PART 12 MobiProg: Adaptive Programming System for Cloud-Enabled Smartphone Applications

- Dero Gharibian, Graduate Student
- Ramesh Govindan, Faculty, PI*
- Michael Gray, Graduate Student
- Nupur Kothari, Staff
- Taehee Lee, Graduate Student
- Todd Millstein, Faculty, PI
- Nilesh Mishra, Graduate Student
- Luis Pedrosa, Graduate Student*
- Jeff Vaughan, Staff

* Primary Contact

Overview

With the advent of Smartphones as an emerging class of personal Internet capable devices, mobile applications, or apps, are rapidly becoming the cornerstone of what defines the user's experience. While many of these apps will perform simple tasks on the phone, a new class of cloud-enabled applications is becoming more popular. These apps augment the smartphone's capabilities, allowing them to leverage the large datasets and the computational power that can be harnessed in large-scale computing infrastructures known as server clouds.

Many of these cloud-enabled mobile applications delegate all or most of the business logic to the cloud, reducing the smartphone to a thin client. However, there is a vast universe of unexplored possibilities where the smartphone collaborates more closely with the cloud to enhance the user's experience. Such applications can be described in abstract as a function \( F \), as shown in Figure 1. This function can depend on time-varying inputs from the phone \( I(t) \), as well as data-sets residing in the cloud \( D(t) \). The MobiProg project aims to create a new framework for the development of such cloud-enabled smartphone applications.

Using this framework, developers implement a single application in a way that is agnostic to running on the phone or in the cloud. Once the application is implemented using this framework, a run-time system partitions the application, automatically executing the relevant portions on the phone or in the cloud so to maximize a given performance metric (e.g. processing time or battery life). This approach not only removes the burden of manual partitioning from the developer but also accounts for situations where the optimal partitioning decision depends on the runtime context, such as radio technology in use (GSM, CDMA, or WiFi) and the amount of charge remaining in the battery.

Approach

The MobiProg framework allows applications to be implemented following a simple data-flow-like pattern, i.e. applications are written as a linear pipe-line of components. Furthermore, since individual components are implemented using the Java language, the Java Runtime Environment automatically handles most issues regarding component-level portability, namely between the server and the Android smartphone platform, our initial development target. By enforcing such constraints on the developer, the partitioning algorithm is drastically simplified, being reduced to making the decision of which components execute locally or remotely. The run-time system then invokes each component on the right platform, automatically marshalling any data to where it is needed.

Given a run-time such as this, the key to executing applications in an optimal way lies in the quality of the decisions made by the partitioning algorithm. Such a decision becomes particularly challenging given the many factors that equate into it. From a high-level point of view, one or more consecutive components should be executed remotely if their local processing cost is greater than that of sending the data to the remote system, executing there, and retrieving the results. Already the decision depends on such dynamics as the kind of radio in use, as this directly impacts the cost of sending and receiving data over the network. However, the harder question is how does one predict the cost of executing a component on a particular platform without actually performing the execution.
Previous frameworks have tackled this issue by making the simplifying assumption that such a prediction can be made simply by averaging the cost of previous executions. While this assumption may suffice for some components, it entirely disregards several key factors such as algorithmic complexity and the influence of the input data on the algorithms resource usage. To account for these factors, MobiProg employs a prediction framework where a set of predictor functions can be leveraged to more accurately derive component costs for each execution and, consequently, make better partitioning decisions.

We have implemented several prediction functions with varying degrees of generality and precision. The simpler functions, for example a linear regression of experimental profiling data vs. input size, may provide less accurate predictions but can operate with virtually any input data type. To further improve this accuracy, these metrics can be combined with application specific heuristics, for example a word counting function to predict the complexity of speech recognition algorithms. Finally, a more general approach that we have also developed tries to retain a high level of accuracy while keeping the generality of the simpler approaches by using static analysis to heuristically determine the algorithmic complexity of the component. Using this final approach, each component's code is statically analyzed off-line, producing a signature indicating its input complexity dependencies. Armed with these signatures, the run-time system can analyze concrete input data and get a quick yet precise estimate of the cost of executing the component with that specific input.

System Description
An initial run-time system has been implemented for both the server platform and the Android smartphone platform. While the current implementation does not yet perform fully automatic partitioning, it does allow us to manually test several partitioning configurations and validate assumptions. A few key applications have also been ported to use this system, namely a speech recognition and translation application, a bar-code scanner, an optical character recognition application, and a Sudoku solver.

The prediction framework has also been implemented with several initial prediction functions. These prediction functions already include some generic heuristics based on input size and average executions times, as well as some application specific heuristics for the speech recognition component (e.g. word counting and signal-to-noise ratio).

Finally, the static analysis based complexity heuristic is under active development and can already accurately detect the complexity of some simple hand crafted test-cases. Additional corner cases are being handled as the system is being tested with publicly available third-party libraries.

Accomplishments
While a full experimental evaluations has not yet been completed, a few initial tests have been conducted on the proof-of-concept system. Preliminary findings seem to suggest that there is in fact room for improvement in static partitioning schemes, as system context and dynamics can have a significant effect in the outcome of the partitioning decision. Experimental profiling data also shows that component complexity, as one would expect, can also vary drastically based on input.

Further experimentation is needed to evaluate the final system, specifically to quantify how much of an improvement, if any at all, can be achieved in relation to statically partitioned applications and state-of-the-art dynamic portioning schemes.

Future Directions
In the upcoming year we expect to finish the development of the proof-of-concept framework prototype and to complete the effort of porting key initial applications. A more thorough experimental evaluation will also be conducted and the work is expected to be published in peer-reviewed venue within this time-frame.