ABSTRACT
As application demands outpace the evolution of battery technology, many smartphone “app” developers will soon explore offloading compute-intensive tasks to the cloud. Such cloud-enabled mobile applications effectively partition application functionality between the phone and the cloud. Application partitioning must be dynamic, to successfully adapt to variability in resource availability. Dynamic partitioning systems rely on the ability to predict an application’s component’s resource usage, for which prior work has used simple approaches. In this paper we propose the use of complexity metrics that enhance these predictions by taking into account relevant properties of each component’s input, both general-purpose (e.g., size) and type-specific (e.g., number of words in an audio sample). Our predictors improve the energy efficiency of partitioning a speech recognition library by 21% or more.

1. INTRODUCTION
As smartphones gain popularity, mobile applications, or “apps”, are rapidly becoming the cornerstone of what defines the user’s experience. Already, there are apps available for various phone platforms that provide useful capabilities like speech recognition, restaurant recommendations, automatic logging of workouts, etc. Many of these capabilities place large demands on smartphone resources. Thus, developing these applications requires achieving a delicate balance that ensures application usability without significantly degrading battery lifetime.

To address this challenge, many developers have started using cloud resources to run compute-intensive applications or libraries. This requires significant programming effort, since developers now have to manually reason, for every application, which application components run on the phone and which on the cloud. Moreover, in many cases, it is not clear if the same partitioning strategy for a given app will work for all network conditions and inputs. Thus, there is a need for a dynamic offloading strategy that takes into account the cost of executing a component on the phone vs. the cloud, and attempts to reduce energy consumption without significantly impacting application performance.

Any such offloading strategy must manage complex tradeoffs. For example, moving a computation to the cloud reduces CPU energy consumption but increases energy use by the networking hardware. In general, the relative costs and benefits of offloading may depend on hardware (how expensive is 3G on this phone?), context (is WiFi available?), and workload (how large is the input image?).

Prior work has designed execution frameworks to help with the problem of partitioning application components between the phone and the cloud [1][2][6]. Some of these [1][2] use runtime profiling to predict the resource usage of a component and hence the cost of executing it on the phone or the cloud, and base their partitioning decision on this. While this is a great first step, these systems use simple predictors, like the average computation cost of a piece of code across all inputs seen so far. These predictors can result in suboptimal performance since the computational resource usage of many components can depend on the input. For example, the resource usage of a speech-recognition system can depend on input complexity metrics, such as input size, or the number of distinct words in the audio sam-
In this paper, we propose to use predictors based on input complexity metrics to improve the performance of application partitioning decisions. We motivate the need for input complexity metrics and dynamic partitioning, using an example in Section 2. We present a framework for resource usage prediction that can be easily extended by users to incorporate new application-specific input complexity metrics (Section 3). We combine this with a simple partitioning framework to build a proof-of-concept remote execution system. We demonstrate that for a speech recognition library, our framework improves performance even with the simple, application-agnostic complexity metric of input size (Section 4).

2. MOTIVATING EXAMPLE

We motivate the need for a dynamic partitioning scheme for smartphone applications that takes into account input complexity, using a real world example. We present here a brief experiment using a speech recognition library based on a port of CMU Sphinx4 [7] to Android. Internally, the speech recognition system can be broken into a linear pipeline of components, as illustrated in Figure 1. Figures 2 and 3 respectively show run times for each of these components, as well as the size of the data transferred between them, when executed for two audio samples of different sizes.

These figures show that the execution time and data transfer size varies greatly with the inputs. For both these inputs, while component 9, the Recognizer, is a clear candidate for off-loading, the optimal point to partition the application is not immediately clear. If the phone were close to a WiFi access-point with great reception (and consequently a low per-bit energy cost), it may very well be worth sending the raw audio data to the server where it can be processed almost instantly. If, on the other hand, the phone were using the cellular network with very poor reception, a more intelligent strategy might be to extract the audio features on the phone and leave only the final recognition to run in the cloud. Finally, this decision of where to partition might vary with the input, since the exact trade-offs might be different for different audio samples. As shown in the figures, different inputs can vary considerably in the per-component processing times and data-transfer sizes they incur.

Hence we can see that, given the significant variability one can expect to encounter at runtime due to various network conditions and different inputs, no single static partitioning can be optimal. Furthermore, since the decision of where to execute a component must be made before its execution, its resource usage must be predicted in order to compare the respective costs of executing it locally vs. transferring the data for remote execution. Since inputs have a significant impact on these costs, these predictors must be a function of the input: in our approach, they depend on pre-determined input complexity metrics.

3. DESIGN AND IMPLEMENTATION

In this section, we present the details of a prediction system that uses a vector of input complexity metrics to predict resource usage of various application components. We also describe briefly our implementation of a basic remote execution framework that we use to demonstrate the utility of this prediction system.

3.1 Prediction System

The goal of the prediction system is to predict component resource usage, so that the remote execution framework can make informed partitioning decisions. An important design goal is to ensure that the system is modular, and easily extensible to new user-defined or application-specific input complexity metrics.

We represent input complexity as a vector $C$ whose components describe metrics. We model a component’s resource use by an polynomial $P(C)$, and for simplicity assume $P(C) = k \cdot C$ where $k$ is a vector of constants. Coefficient vector $k$ is learned automatically using test inputs during an initial training phase. In practice, both the complexity metrics and the test inputs would be specified by the user. We believe this does not represent a large burden on developers, since the metrics can likely be defined on a type-specific basis (e.g., audio, video, etc.) and shared across many components that accept
the same type of data as input. Furthermore, developers already test and debug their applications on test inputs.

To train or use predictors, a developer must specify a metric function $M$ that computes complexity metric scores from inputs. Ideally, metric functions should be lightweight, and the developer may need to balance metric utility with overhead. The predicted cost of running a component is naturally computed by

$$cost = P(M(input)).$$

Defining $M$ and $P$ independently allows application developers to create type-specific complexity metrics that extract any relevant information from inputs, while also allowing any algorithm-specific dependencies and platform-specific resource costs to be captured in the polynomial coefficients learned during the training phase. Furthermore, type-agnostic input complexity metrics, such as input size and entropy, can also be used for opaque data types where the developer has not provided an otherwise more relevant metric. We have implemented this prediction system within a simple remote execution framework that we describe next.

### 3.2 Remote Execution Framework

To aid in testing the utility of input complexity metrics for building efficient cloud-enabled smartphone applications, we have developed a basic remote execution framework that allows for dynamic partitioning. While this framework, in itself, is not the focus of this paper, a brief explanation of how it operates is given for context. This framework, based on message passing between state-less components, can be used to develop both self-contained libraries, as well as entire applications. The main focus of this effort is to allow functionality to be implemented in a platform agnostic fashion, such that it can be automatically and transparently migrated between the phone and the cloud. Given such a system, the developer needs only to implement the library or application as a set of interconnected components and the framework runtime automatically decides how to best partition its execution and transparently marshals data and executes the appropriate component where it is needed. In its current implementation, both the framework and the components are implemented in Java and the runtime runs on a standard J2SE virtual machine on the server and in the Dalvik virtual machine on the Android smartphone platform.

The remote execution framework uses a simple algorithm to make partitioning decisions. The purpose of this algorithm is to find a partition that globally optimizes the application’s total cost according to some user specified metric (run time or energy), taking into account both the cost of locally executed components and the cost of marshaling data to and from the cloud for remotely executed components as computed by the prediction system. In our framework, this algorithm is implemented online, in the sense that the partitioning decision can be revised after executing each component, using the new information produced to provide more accurate predictions.

Although there are multiple possible partitioning algorithms, depending on the component graph structure and the kind of predictions that are to be relied upon, we use a simple greedy approach that at every point in time tries to optimize the cost of handling the next component a message is destined to. While this algorithm is simple and light-weight, it may lead to sub-optimal decisions since it does not examine more than one component in the future. However, we find during our experiments, that for our specific speech recognition application, even this simple algorithm provides good results.

### 4. EXPERIMENTAL EVALUATION

In this section we evaluate the effectiveness of the proposed system through a series of controlled experiments.

We ported the CMU Sphinx4 speech recognition library [7] for use with our remote execution framework, componentizing its internal functionality as shown in Figure 1. The component diagram represents a linear pipeline. In any optimal partitioning, either the entire pipeline will execute on the phone, or a contiguous chunk of it will execute in the cloud. We built a simple Android application that given an audio sample, uses our Sphinx4 port to perform speech recognition, and return in text form, the words in the audio sample.

We conducted experiments using an Android Nexus One phone. We used a simple input size complexity metric for prediction. We first trained our prediction system using a corpus if 10 short audio samples. We then ran the speech recognition application on the Nexus One phone for 10 different audio samples, collecting detailed logs that allowed us to determine for each execution cycle, the predictions made, the partitioning decision made, and finally, the cost of execution of each component (measured both in energy and time).

Figures 4, 5, and 6 evaluate the accuracy of the predictions for three different quantities: energy consumption, execution time, and output size. Even though we used a generic input size complexity metric, the predic-
Predicted and Actual Energy Consumption Scatter Plot

Figure 4: Predicted and Actual Energy Consumption Scatter Plot

Predicted Run Times (s) vs. Actual Run Times (s)

Figure 5: Predicted and Actual Run Time Scatter Plot

Predicted Output Size (MBytes) vs. Actual Output Size (MBytes)

Figure 6: Predicted and Actual Output Size Scatter Plot

Total Energy Costs for Different Partitioning Strategies

Figure 7: Total Energy Costs for Different Partitioning Strategies

tions do show a rather strong correlation (0.74, 0.73, and 0.99 respectively for energy consumption, execution time and output size), despite some noise due to system dynamics. Indeed, for the output size predictions (the only truly deterministic of the three predicted metrics), the prediction is near perfect.

Finally, we conducted more experiments to evaluate the optimality of the remote execution system as a whole. We ran the speech recognition application on the phone using both the WiFi and the 3G radios, each of which has a different energy profile, and speed. We measured the average cost of performing a full execution cycle and compared it for the following partitioning strategies.

- **All-Phone**: All of the components were executed on the phone.

- **Expert**: The developer made an educated guess as to which static partitioning would be the best. All audio processing and feature extraction was performed on the phone, leaving only the final Recognizer component to be executed remotely.

- **Optimal**: A post facto offline analysis was performed to determine which would have been the optimal partitioning, emulating an omniscient partitioner.

- **Greedy**: The previously described greedy partitioner was used.

The data shows that, even using a partitioning algorithm as simple as the greedy approach, significant savings can be achieved as compared to the base-line All-Phone scenario (95% and 21%, respectively for the WiFi and 3G scenarios). Indeed, for the WiFi scenario, the greedy strategy managed to find the optimal partitioning while, when using the 3G radio, the solution found was nearly identical to the expert partitioning. This sub-optimality is a consequence of the shortsightedness of the greedy algorithm and the high initial cost of using the 3G radio. Under the circumstances, the algorithm continually chooses local execution in an effort to avoid using the radio entirely. These decisions backfire as ultimately, the system recognizes that the final Recognizer component is too expensive and decides to offload.

5. RELATED WORK

Offloading computation from smartphones and mobile devices in order to reduce their energy footprint, as well as improve performance, has been explored along various dimensions. Researchers have considered partitioning the code executing on the device, and offloading parts to a remote server in an effort to conserve energy at the mobile device, as well as to improve performance.

Kremer et al. [3] propose a remote execution framework for mobile devices, where the partitioning decisions are made statically, and the code can be partitioned into at most two parts, i.e., the computation cannot return to the mobile device. SpatialViews [5] is a programming framework designed for mobile adhoc networks...
that requires the users to manually specify static partitions as part of the program. Both of these systems are geared towards specifically reducing energy consumption, rather than improving performance. Unlike, these works, we believe that the decision of where to partition should be made dynamically, since it depends not just on the network conditions, but also on the complexity of computation which may vary with different inputs.

Many works dynamically decide how to partition and offload computation. MAUI [2] and Wishbone [4] are two systems which profile the energy usage and computation costs of various components of a program in order to decide at runtime how to partition the computation. MAUI is an execution framework designed for smartphones, while Wishbone is a framework to build sensor network applications. In Chroma [1], users provide a set of “tactics”, each of which is a different way to implement a mobile application. The runtime chooses among these depending on current conditions, and decides whether to offload each function in the chosen tactic, using runtime profiling of resource usage for that function. None of these works consider the effect of input complexity on the cost of a component’s execution. The authors of Odessa [6], a framework for building interactive perception application for smartphones, recognize that the complexity of computation depends on the inputs. They leverage this in the context of their specific class of applications, along with runtime profiling to make offloading decisions dynamically. We propose going a step further and predicting the complexity of an algorithm and its resource usage to make partitioning decisions without the overhead of runtime profiling.

6. CONCLUSION

In this paper we made the case for an automatic partitioning system for cloud-enabled smartphone applications that uses input complexity metrics to estimate resource usage. Current state-of-the-art partitioning algorithms use simplistic approaches to estimate a component’s resource usage, such as using the average value of previous executions. However, a component’s actual resource usage also depends on the input, unaccounted for in these simple models. By estimating resource usage using generic and application-specific complexity measures on each component’s input, in conjunction with automated learning procedures to assimilate each algorithm’s and each platform’s inherent complexity costs, more accurate predictions can be achieved, allowing the partitioner to operate in a more educated fashion.

We showed experimentally that using simple heuristics based on input size, resource usage can be predicted with a correlation of 0.73 or more. Furthermore, using this information, a simple greedy partitioning algorithm was able optimize a speech recognition library’s execution, saving 21% or more on power consumption. While the initial results shown here are promising, more work is required to further improve the system. Better prediction accuracy can be achieved by using more application-specific heuristics. These will naturally be developed as we port more applications to use our system. As for the partitioning algorithm, while the greedy approach illustrated here did perform quite well, other algorithms (based on dynamic programming and linear programming, for example) may be able to achieve better results. These efforts, as well as other usability enhancements to the framework, are left for future work.

7. REFERENCES