# Purdue University Purdue e-Pubs

**ECE Technical Reports** 

**Electrical and Computer Engineering** 

5-20-2011

# Enabling Automatic Offloading of Resource-Intensive Smartphone Applications

Abhinav Pathak

Y. Charlie Hu

School of Electrical and Computer Engineering, Purdue University, ychu@purdue.edu

Ming Zhang

Paramvir Bahl

Yi-Min Wang

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Pathak, Abhinav; Hu, Y. Charlie; Zhang, Ming; Bahl, Paramvir; and Wang, Yi-Min, "Enabling Automatic Offloading of Resource-Intensive Smartphone Applications" (2011). *ECE Technical Reports*. Paper 419. http://docs.lib.purdue.edu/ecetr/419

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

## Enabling Automatic Offloading of Resource-Intensive Smartphone Applications

Abhinav Pathak Y. Charlie Hu Ming Zhang Paramvir Bahl Yi-Min Wang

TR-ECE-11-13

May 20, 2011

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

## **Contents**

1	Introduction	1							
2	Related work								
3 Problem context									
4	Profiling Techniques  4.1 Identifying remotable methods	5 6 6 7							
5	Implementation         5.1 Tracing system & application events          5.2 Partitioning application for runtime offloading	<b>8</b> 8 9							
6	6.2 Experimental setup  6.3 Overall results  6.3.1 Event trace details  6.4 Regression results	10 10 11 12 13 14							
7	7.2 Offloading decision accuracy under varying network bandwidth	14 15 15 16 16							
8	Conclusion	17							

**Abstract** – The limited capability and energy constraint of smartphones have posed a significant challenge to running the "newest and hottest" applications which are becoming increasingly resource demanding, *e.g.*, real-time image recognition. In this paper, we revisit the decade-old general concept of offloading computation to remote servers by focusing on a largely unsolved problem: how to automatically determine *whether* and *when* a smartphone application will benefit from offloading? This is an especially relevant and challenging problem today as (1) modern mobile applications tend to have complex interactions with users and advanced capabilities (*e.g.*, GPS and camera) and hence cannot be offloaded as a whole; (2) whether an application component, *e.g.*, a method call, will benefit from offloading depends on its execution time on smartphone and the size of state to be shipped, which in turn depend on the input parameters.

We present the design and implementation of XRay, an event-tracing-based profiling tool that identifies methods in a smartphone application that can be offloaded to a remote server, and determines whether and when offloading the methods will benefit the application. Our experiments of applying XRay to a set of smartphone applications show that after a small number of offline profiling runs, XRay can automatically generate offloading decision logic for each remotable method that makes correct offloading decisions in future online executions of these applications under a priori unknown input parameters and network conditions.

## 1 Introduction

Smartphones have limited capabilities compared to their desktop counterparts. The CPU and memory of the newest high-end smartphones significantly lags behind those used in desktop and server PCs and this discrepancy in capabilities is unlikely to go away anytime soon. Additionally, smartphones are severely constrained by their battery. New desirable features such as GPS, camera, high-resolution color screen, and high-speed wireless interface increase the demand on smartphone's battery, which unfortunately is not following Moore's Law. However, consumers expect similar experiences on their smartphones as they get on their desktop PCs. Today, running resource-intensive applications such as image recognition, document translation, information retrieval, and real-time gaming on smartphones is either intolerably slow or infeasible.

One popular solution that has attracted the attention of researchers for over a decade is to offload part of, or migrate the whole computation to remote servers. This solution looks increasingly appealing because lots of server resources are available and wireless technologies, such as LTE-A, WiMAX and high speed Wi-Fi (IEEE 801.11n) are becoming fast and ubiquitous. In fact the recent world-wide availability of large-scale cloud computing infrastructure from Amazon, Google, Microsoft, and Yahoo has spurred the growth of a variety of mobile applications which leverage resources in the cloud [1].

Existing approaches to offloading computation generally fall into two broad categories: virtual machine (VM) migration and application partitioning. Recent examples in the first category include CloneCloud [2] and Cloudlet [3]. Both use VM migration to transfer soft and hard states between the mobile device and the infrastructure server. Since a VM encapsulates potentially relevant states on a mobile device, there is no need to understand or modify the applications. The downside is that traditional VM migration incurs high shipping overhead and does not work for the emerging class of applications which heavily rely on various sensors on mobile devices. Examples in the second category include Chroma [4, 5], Spectra [6, 7], and Maui [8] where applications are explicitly written for local and remote executions. While this approach incurs less offloading overhead, it usually involves various degrees of manual program restructuring effort. For instance, Spectra [6, 7] and Chroma [4, 5] ask developers to specify meaningful ways to partition an application. Maui [8] requires programmers to identify and mark code regions that can potentially be offloaded.

To overcome these shortcomings and limitations, we present XRay, a tool that enables automatic partitioning and offloading of existing resource-intensive smartphone applications.

There are several challenges in automating the offloading of an existing smartphone application. First, not every component of an application is amenable to offloading. Consider a simple example of a user playing a

chess game against a machine. The chess program iterates between the user's move and the machine's move. The former requires mouse clicks by the user and has to be processed locally on smartphone. The latter requires CPU computations only and is a good candidate for offloading. Hence the first challenge is *how to automatically identify the remotable components of an application*. Second, even if a component of an application is remotable, offloading it may or may not be worthwhile depending on the offloading cost and benefit. The latter is usually determined by various factors such as the amount of computation, the size of offloading states, and the available network bandwidth. Thus the second challenge is *how to accurately quantify the cost and benefit of offloading a remotable component on-the-fly*.

We have developed a tool called XRay to meet both challenges. XRay traces all the relevant system-level and application-level events during application execution. We classify these tracing events into *local* ones (*e.g.*, GUI, GPS, and sensor) and *remotable* ones (*e.g.*, CPU and memory), and identify *remotable methods* as those that never trigger any local events. XRay derives simple yet effective models during offline profiling runs of an application to characterize the execution time of a remotable method on the smartphone, the serialization/deserialization time of that method, and the size of the state to be transferred over the network for remote execution, as functions of the inputs to the method. The models thus constructed by XRay are then embedded into the application running on the smartphone to make offloading decisions during future online executions of the application. Since the offloading cost and benefit of a method may change across different call instances, XRay employs linear regression analysis in constructing the models which can accurately predict the offloading cost and benefit given any inputs to the method. We have implemented XRay on Windows Mobile 6.x (WM6).

To make use of the offloading decision logic produced by XRay, we implement an automated scheme that partitions any existing .NET application by creating RPC wrappers around the remotable methods and inserting the offloading logic to perform opportunistic offloading of remotable methods on-the-fly to optimize the execution time. We apply XRay to several real smartphone applications, including barcode reader, skin detector, pdf2text convertor, search indexer, and chess. Under a variety of inputs and network conditions, the offloading logic generated by XRay makes near-optimal decisions compared to an ideal scheme which assumes prior knowledge of the cost and benefit of every call instance. Moreover, such correct offloading decisions led to dramatic reduction in execution time (25% to 89%).

We believe XRay represents a significant step forward in enabling thousands of existing resource-intensive smartphone applications to benefit from cloud computing resources. Further, as cloud computing infrastructure becomes commonplace, XRay should prove tremendously useful in assisting application developers in developing future "cloud-aware" applications, *e.g.*, guiding the application developers in structuring the application code to maximally exploit the benefits from offloading computations to the cloud.

The rest of the paper is organized as follows. After discussing the related work in §2, we set the problem context of our work in §3. §4 presents our approach to identifying remotable methods and estimating offloading cost and benefit using event traces. We present the implementation of XRay on WM6 in §5. §6 presents the profiling results of a set of real smartphone applications as well as the profiling overhead. §7 evaluates the effectiveness of XRay-based offloading scheme under different application inputs and network conditions. Finally, §8 concludes the paper.

#### 2 Related work

Application partitioning and remote execution have been extensively studied in mobile computing research. Spectra [6, 7] dynamically balances energy consumption with user metrics. It continually monitors application resource usage and uses this information to decide between local and remote executions. Chroma [5, 4] semi-automatically partitions an existing application by taking advantage of application specific knowledge. It takes as input all the meaningful partitioning strategies of an application, specified by application developers in a declarative form, and selects an appropriate partitioning strategy according to online resource demand prediction.

Maui [8] is a recent system that supports fine-grained C# .NET code offloading to save energy on smartphones. It requires the developer's assistance in marking remotable methods and does not explicitly model the impact of input variables on execution time, transferred state size, or energy usage. Compared to the approaches above, XRay can automatically extract remotable methods and precisely compute offloading cost and benefit at runtime without any developer's help.

There is a large body of prior work on application partitioning in the context of sensor networks and distributed systems. Wishbone [9] takes a profile-based approach to automatically partition data streaming sensor applications. It aims to jointly minimize network and CPU load by solving an integer linear programming problem. Coign [10] automatically partitions applications written in the COM framework without instrumenting the source code. Its goal is to minimize the communication overheads among different machines. These approaches cannot be directly applied to mobile applications.

Cyber-foraging [4] and data staging [11] refer to the use of nearby surrogate computers to improve the performance and capabilities of mobile systems. Goyal and Carter [12] proposed a similar idea to reduce the response time of resource-intensive mobile applications. More recently, Slingshot [13] replicates application state on both surrogate servers and remote servers to enhance the fault tolerance of replicated applications. However, they all assume the applications have already been partitioned properly.

Application profiling has drawn lots of attention in various research efforts. Magpie [14] leverages ETW [15] tracing in Windows PC operating system to track the resource usage of applications by correlating relevant messages. Along similar lines, Yuan *et al.* applied learning techniques to application event traces for automatic fault detection and diagnosis [16]. Quanto [17] is a profiling tool for tracking network-wide energy usage of embedded applications. It is useful for diagnosing hardware energy leaks and performing energy-aware scheduling. None of these profiling tools are designed for offloading smartphone applications. Our work is the first to use system call tracing to offload smartphone applications.

#### 3 Problem context

Today, a rich set of advanced features on smartphones have enabled many sophisticated mobile applications. An application may interact with users through keyboard and touch screen. It may provide customized service based on user location information obtained from GPS. Of course, every application has to rely on memory and CPU. Not all applications lend themselves to offloading. For instance, a GPS-based "where am I" service may be better off running locally. A majority of current day smartphone applications perform simple tasks, like display weather, post a wall-update, check email, tweet etc. A cardinal reason for resource-intensive applications to not exist today in large numbers is the limited capabilities of smartphones, which significantly increases the runtime and energy consumption of these applications. This causes annoyance among application users and degrades the developer's reputation. For *e.g.*, a sudoku solver application faced severe criticism from its users when they generated medium and harder difficulty level puzzles. As smartphone market grows, the demand for such applications would grow since some of these applications are basic requirements (like antivirus, search indexing, voice recognition, etc.).

Application developers currently solve this challenge in two ways, (a) sacrifice complicated algorithms in building the application (followed by several mobile antivirus companies, mobile voice recognition softwares, etc.) (b) completely give up the functionality (*e.g.*, Mobile chess games do not offer higher difficulty levels in game against computer). Our work focus on these kind of resource-intensive applications, which are currently rare due to limited capabilities of smartphones, but we believe it to be necessary for application developers to provide them in near future.

There are two primary challenges in offloading a resource-intensive application. First, an application typically comprises a variety of functional components. For instance, an image recognizer will have a GUI component to interact with user, an I/O component to read/write files, and an algorithmic component to perform image recognition. Some components (*e.g.*, GUI) should run locally on the smartphone while others (*e.g.*, algorithmic) could

potentially run on a remote server. We must partition the application in an appropriate manner before it is ready for offloading.

Second, offloading an application component induces not only benefit but also cost. We should offload a remotable component only when the benefit outweighs the cost. The actual offloading cost and benefit depend on various factors including the size of the application state that needs to be transferred, the available network bandwidth, and the processing time of the component on the smartphone. Making things even more complicated, these factors are not static; application state and processing time may change under different inputs. The network bandwidth may also vary across different times. The correct decision needs to be made at runtime on whether to offload a remotable component.

Existing approaches to offloading applications often require various degrees of manual work by application developers [4, 5, 6, 7, 8]. Such approaches can be undesirable due to a number of reasons. A resource-intensive application can be quite complex, comprising thousands of lines of code and hundreds of methods, with a plethora of dependencies between different methods and objects. Manually partitioning such an application for offloading is not only time-consuming but also error-prone for programmers. Sometimes, the entire application or certain portion of the code is written by a third-party, making it even more difficult to reason about. Given the large number of existing resource-intensive applications which can potentially benefit from offloading (*e.g.*, image recognition, gaming, document conversion, *etc.*), an automatic tool will significantly reduce the burden imposed on application developers.

One compelling way to automate application partitioning is to perform static code analysis. One could develop a parsing tool to extract the remotable components (*e.g.*, those that do not interact with GUI, GPS, or sensors) from the source code. However, one possible problem with this approach is that some applications use third-party binary libraries. The parsing tool cannot tell whether the components that use the binary libraries are remotable or not. Moreover, the parsing tool is unhelpful in estimating the offloading cost and benefit of a component, which are likely to change on-the-fly.

Another potential approach for offloading could be through transparent checkpointing the state of running binary. One could develop a tool to profile runtime binary on mobile, freeze its execution at a particular point, transfer its state (memory, stack, registers, PC, open files, etc.) to remote powerful server, resurrect its state, complete the compute intensive part and follow the cycle to bring it back to mobile. This approach does not require source code of the program. We initially followed this path, but realized that such a tool requires advanced API support from OS to perform the steps listed above. Specifically, the ability to control a running process's memory is a core pre-requisite in building this tool. We found that the kernel of windows mobile 6.5 (OS of phone we use for evaluation) do not support these advanced APIs. As a result, we drop back to an approach where we require source code of the application.

**Our Approach.** To tackle the problems above, we develop a tool called XRay based on event tracing to enable automatic partitioning and offloading of resource-intensive smartphone applications. XRay traces all the system-level and application-level events that need to be handled locally on the smartphone, which are then used to identify remotable methods. It also keeps track of the usage of various resources related to the offloading of application methods, including CPU and memory. The only domain knowledge needed to use our tool, is the potential set of inputs for the application. The user needs to supply a diverse set of inputs to our tool. To use the tool, given an existing application, we first use XRay to produce its *profile* by running it with a training set of different inputs. The profile includes all the remotable methods and a set of linear functions, each of which captures the relationship between the inputs to a method and the offloading cost and benefit of the method. Afterwards, we invoke a partition tool that automatically partitions the application by creating RPC wrappers and embedding offloading decision logic based on the profile around each remotable method. In future online executions, the partitioned application will opportunistically perform method call offloading to optimize the total execution time on the smartphone. We describe each of these steps in detail in the subsequent sections.

We argue that the domain knowledge (about diverse input set) required to use our tool is relatively easier to

procure and can even be obtained from a non-professional application user. Compared to earlier work [8], this is a substantial improvement, since previously only the programmer of that specific application can provide the in-depth information required to partition the program. In cases where third party closed source DLLs are used to develop an application, even the programmer can not tell which methods are remotable.

## 4 Profiling Techniques

In this section, we describe the two key techniques in XRay. We will present the implementation details in the next section. As just mentioned, there are two challenges in offloading a resource-intensive smartphone application. We first explain how to extract remotable methods from application source code. We then describe how to compute the offloading cost and benefit of a method call instance.

## 4.1 Identifying remotable methods

As in prior work [8], the offloading opportunities in running an application are identified at the granularity of methods. This has proven to provide sufficient offloading flexibility while simplifying the process of extracting the application variables required for offloading. A method may use various types of resources on the smartphone. We classify different types of resources on a smartphone into *local* and *remotable* ones based on whether they can be replicated on a remote server or not. In this paper, we only consider CPU and memory as remotable resources. Other types of resources, such as keypad, network interfaces, I/O, GPS, *etc.*, are considered as local ones. If any of these resources used by a method (or any methods called by it) is local, we classify the method as a *local* method, *i.e.*, it should not be offloaded. Otherwise, we classify it as a *Remotable* method (RMethod). The resource usage of each method can be tracked via system call tracing at runtime (§5.1).

We note the above classification of resources into local and remotable can change under different replication policies. §7 shows that even such a conservative replication policy can benefit a variety of CPU-intensive applications. We could also replicate the file system on a remote server. This would allow I/O intensive applications to exploit the benefit of offloading as well. Network calls can also be offloaded using different techniques, like tunneling, source address spoofing (to ensure TCP does not break), etc. We leave for the future the task of extending our work to support file system and network call offloading.

Now given the source code of an application, we first annotate each method with two unique application-level events at the entry and exit points. Next we compile the annotated source code and run the annotated application while logging all the system-level (via an automated kernel logging tool described in §5) and application-level events. From the event trace, we can easily find the start and end of each method call instance. We determine a method to be remotable if none of its call instances contain any local events, *i.e.*, events that use local resources. If a method is never called during the execution run, it is classified as local.

We discuss two possible complications that may arise. First, multithreaded applications need special handling. In particular, shared memory is considered as a local resource. This is because if shared memory is replicated, it may be accessed simultaneously on the smartphone and the remote server. However, an application does not use system calls to access shared memory. To track the usage of shared memory, we leverage the observation that accesses to shared memory are properly synchronized with synchronization primitives, *e.g.*, lock(), and instrument the application source code to generate an app-level local event corresponding to each of such synchronization primitives. Logging such new app.-level events can be achieved via CeLog API on WM6 (§5.1).

Second, a method can have many distinct execution paths. Due to the discrepancies between the profiling and future executions of the method, we may mis-identify a local method as a "remotable" one. To tackle this problem, we attempt to exercise the commonly-used execution paths by feeding different inputs to the application during the profiling runs. As long as the method is "remotable" in all the profiling runs, we deem it as "remotable" in common cases, and XRay will attempt to offload the method during online executions if it predicts offloading will

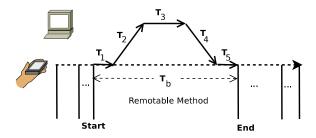


Figure 1. Five steps in offloading process

be beneficial. If a rare, local execution path is indeed exercised while the method is running on a remote server, the offloading recovery mechanism described in §5.2 is triggered to handle the exception.

### 4.2 Computing offloading cost and benefit

We should offload a remotable method call instance during runtime only if it reduces the running time or energy consumption on the smartphone. In this paper, we use running time as the metric to make offloading decisions. §7.4 shows that running time savings often lead to energy savings as well. Our methodology still applies if we use energy as the metric.

#### 4.2.1 The Basic scheme

Given a remotable method call instance, offloading it should reduce its execution time on the smartphone  $(T_b)$ .  $T_b$  can be directly measured using event tracing. To estimate the execution time with offloading  $(T_c)$ , we break down the offloading process into 5 steps as illustrated in Figure 1:  $T_1$ ) suspending application execution on the smartphone;  $T_2$ ) transferring application state to the remote server;  $T_3$ ) executing the call instance on the remote server;  $T_4$ ) retrieving application state from the remote server;  $T_5$ ) resuming application execution on the smartphone.  $T_c$  is the summation of  $T_1$  through  $T_5$ . The call instance should be offloaded if  $T_c < T_b$ .

 $T_1$  comprises the time to find all the variables used by the method and the time to serialize these variables for network transfer. We simply add a few lines of code at the method entry point to find and serialize all the relevant variables, and measure  $T_1$  in a profiling run. We will explain how to find the variables used by a method in §5.2. To estimate  $T_2$ , we record the size of the variables after serialization (ssize). Later on during runtime, we measure the current available network bandwidth (bw) and estimate  $T_2 = \frac{ssize}{bw}$ . Given the huge performance gap between the smartphone and the remote server, we can effectively ignore  $T_3$  in our computation (we acknowledge that in few cases  $T_3$  it could become significant (like shared/slow remote server). This can also be modeled with additional information (like remote serverload, configuration, etc)). This is currently out of scope of the paper). Finally,  $T_4$  and  $T_5$  can be estimated in similar ways as  $T_2$  and  $T_1$  respectively.

During runtime, we measure the current available network bandwidth bw by periodically transferring a 40 KB file between smartphone and remote server. The file size is chosen since it represents typical size of offloading state. In practice, we find this simple method is accurate enough for making offloading decisions and consumes little energy. When ssize is small, we can under-estimate network transfer time using  $\frac{ssize}{bw}$ . This is because TCP cannot fully utilize the available bandwidth during slow start. However, since network transfer time will only be an insignificant fraction of  $T_c$  in this case, such estimation error should have limited impact on offloading decisions.

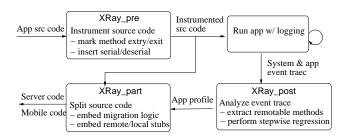


Figure 2. Components and usage of XRay.

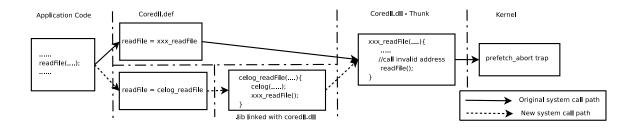
### 4.2.2 Dealing with varying inputs

The offloading cost and benefit of a method usually change with the values of inputs to the method. Since it is impossible to explore all possible input values during profiling runs, we need a way to predict the runtime offloading cost and benefit based on the profiling results measured in a finite number of profiling runs. This is the point where we require domain knowledge about the application. A right set of input is a diverse set consisting of inputs which are small; medium and large as per the application. The basic idea is to construct a function, during profiling runs, that captures the relationship between input variables and each cost/benefit factor, e.g.,  $T_b$ ,  $T_1/T_5$ , and ssize. These functions should be both accurate and simple enough for automatic construction. We observe that a cost/benefit factor often heavily depends on the input size. For instance, in the barcode reader application, it takes much more time  $(T_b)$  and memory (ssize) to recognize an image as the image size increases. Larger memory usage also leads to longer (de)serialization time  $(T_1/T_5)$ . In fact, we have observed similar behavior for a few other applications, including image recognizer and search indexing.

We use regression analysis to construct a model that predicts each offloading cost/benefit factor  $(T_b, T_1/T_5,$  and ssize) as a function of input variables to an RMethod. We first obtain a set of training data by running the application n times with different inputs. We then attempt to find a function which has the best fit in the least-squares. There are many different types of fitting functions, e.g., polynomial and piece-wise linear. In practice, however, we find linear functions work quite well for the applications we studied (§6.1). The choice of training set size (n) reflects the trade-off between profiling overheads and prediction accuracy. A larger n allows us to construct a more accurate fitting function but leads to higher profiling overheads. We will study the effect of n in §7.

A method may have multiple input variables, many of which either have little impact on a cost/benefit factor or strongly correlate with other input variables. We should remove the irrelevant and redundant variables to improve the robustness and accuracy of regression. We use stepwise regression [18] to select the most relevant variables from all the input variables. We start with an empty variable set  $\Phi$ . At each step, we add one remaining variable to  $\Phi$  which most significantly reduces the least-squares error of the regression. This process terminates when there are no remaining variables or none of the remaining variables can significantly reduce the regression error.

We note that the assumption that offloading cost/benefit depend on input size does not hold for all application methods. Using Chess as an example, the MachineMove method takes the current chessboard as input and computes the next move. While the chessboard size remains the same, the computation time  $(T_b)$  can vary dramatically. In such cases, although XRay cannot automatically construct a model to predict  $T_b$ , it can still help to narrow down and profile each RMethod. Currently, XRay makes offloading decisions for such a method based on its average case, e.g., using its mean computation time during all the training runs.



**Figure 3.** System call instrumentations to log additional events

## 5 Implementation

Figure 2 illustrates the component programs of XRay and the overall process of using XRay to partition an existing application to enable runtime offloading. First, we run a pre-processing program (XRay\_pre) which takes the application source code as input and adds serialization and deserialization to the entry and exit points of every method. Second, the instrumented application is executed multiple times with different inputs during which the system and application events are logged via kernel logging. Third, we run a post-processing program (XRay\_post) which analyzes the event trace to identify the remotable methods and construct the regression functions for each remotable method. Finally, we run a partitioning program (XRay\_part) which partitions the application into mobile code and remote server code by embedding offloading decision logic and offloading stubs to each remotable method. We have explained Steps 1 and 3 in the previous section. The pre-processing program contains 500 lines of code (LOC) integrated with a C# parser [19]. The post-processing program is implemented as an 1000 LOC program in VC++ to decode event logs generated by XRay, several R [20] scripts for linear regression, and many perl scripts for parsing output. In the following, we will describe how XRay performs event logging in Step 2 and how an application is partitioned in Step 4.

## 5.1 Tracing system & application events

The system and application events required by XRay to derive application profile need to be logged in the kernel and hence the implementation is OS-dependent. In this paper, we describe our implementation on WM6, though such framework has been available on desktop PC [21, 15].

**Existing tracing support.** We use CeLog [22] to log system wide events on WM6. CeLog is system wide event tracking tool that logs a pre-defined set of events in kernel and other dlls in windows mobile. In essence, CeLog provides a logging function that can be called from anywhere in the kernel or in the application. By default [22], WM6 already uses CeLog to track the events related to CPU utilization by logging context switches, memory utilization, etc. To minimize overhead, the logging function records a small amount of information for each event, which includes the current timestamp in microsecond granularity and a customizable message specific to the event. The event is attributed to the most recent process-thread that was scheduled to run.

**Logging additional system calls.** Presently, WM6 does not log system calls related to many "local" resources (*e.g.*, file system, network interface, GPS, GUI), which are required for distinguishing between remotable and local methods. To log these additional system call events, we instrument two critical kernel DLL files (coredll.dll and ws2.dll) which export most of the system calls.

Instead of through the software interrupt in the ARM processor (SWI), WM6 implements system calls through a process called *thunking* A system call results in a call to a special invalid address. When an such an invalid address is called, a prefetch-abort trap handler is issued to handle the trap. The trap handler recognizes which system call was being made using the numerical value of the invalid address being called. The upper path in Figure 3 illustrates the path of a readFile() call through coredll.dll. This is similar to that of a networking call through

ws2.dll. After readFile() is called, it is renamed to the thunk name xxx\_readFile in coredll.def. Coredll.dll implements a small wrapper for the thunk to call the special invalid address. Finally, the kernel will handle the system call.

The system calls in coredl1.dl1 (file system and GUI) and ws2.dl1 (networking) are slightly detoured through our functions to support logging (lower path in figure 3). Essentially, we replace the thunks (e.g., xxx\_readFile) with our own functions (e.g., celog\_readFile). The new functions use the CeLog API to log the particular system call event with input parameters or return values, and then calls the thunk (invalid address) exported by coredl1.dl1. These new functions implemented are compiled as a a library which is statically linked with coredl1.dl1 during compilation. We use a similar way to log system calls to GPS, GUI, Network and other sensors.

An alternative way to intercept system calls is by replacing system call handlers with our own functions [23, 24]. This approach leverages the setkmode call exported by WM which permits any application to acquire kernel privileges. However, this approach can only intercept system calls exported by coredl1.dl1 which are insufficient for identifying remotable methods. Moreover, setkmode will be removed from WM due to security concerns [25].

## Customizing kernel image on smartphone.

Even a minor change in one of the core libraries in a smartphone OS requires compiling the entire image and flashing it to ROM (Just overwriting the modified coredll.dll on a running phone is not possible). We used platform builder [26] and shared WM kernel source code [27] to build a WM6 image with our modifications. The customized image works directly a WM emulator which runs on a desktop PC. We also flashed a smartphone handset with the customized kernel image. We use this handset in our evaluation. We briefly describe the flashing process as follows.

A WM6 kernel image consists of four sections: boot sector, eXecute In Place (XIP), IMGFS (image file system), and an optional FAT section. The XIP contains the kernel image NK.exe and about 17 DLLs (including coredl1.dl1), which form the core of WM6 kernel. These DLLs are never loaded into RAM but instead are directly executed from ROM. This is why the section is called eXecute In Place (XIP). As a result, the base addresses of the DLLs in XIP are hard-coded. The IMGFS section contains ws2.dl1 and the OEM files that are commonly seen on a fresh WM installation. To flash the ROM of a smartphone with an emulator image, we simply copy ws2.dl1 from the IMGFS of the emulator image to the IMGFS on the phone. A similar procedure cannot be followed for coredl1.dl1 since the hard-coded addresses in the XIP of the emulator image are different from those on the phone. Instead we have to re-calculate all the hard-coded addresses and copy the entire XIP from the emulator image to the phone.

#### 5.2 Partitioning application for runtime offloading

After obtaining the profile of an application which includes its remotable methods and the regression functions of each remotable method, we partition each remotable method into mobile and remote stubs to enable profile-based offloading at runtime. We leverage the techniques developed in Maui [8], a recent effort for building code offloading of C# .NET applications.

To offload a method, we need to identify the set of objects/variables that need to be transferred between the smartphone and the remote server. The problem gets complicated in case of pointers as determining the size of the referenced memory is non-trivial. Hence, we only consider the applications written in C# .NET managed code (no pointers) in our current implementation. Ideally, the objects/variables related to a method include the method arguments, return variables, and class or global objects/variables that are accessed by the method itself or by any methods it calls. For simplicity, we offload *all* the global objects/variables, objects/variables belonging to the class which contains the remotable method, method arguments, and return variables. We use a C# parser [19] to extract such information from the source code. In §7, we show that applications can still benefit significantly from

offloading even though the size of the transferred state may be bigger than the ideal size.

XRay generates two stubs for each remotable method, one for the smartphone and one for the remote server. XRay encodes the offloading decision logic into each mobile stub for it to determine whether to offload the remotable method according to the current estimate of offloading cost and benefit, taking into account the sizes of relevant parameters. If it decides to offload the method, it serializes the method and its related state (described above) and sends the stream to the remote server and suspends the current thread waiting for communications from the remote server. We utilize the XML serialization feature of the .NET framework as it is supported on both the smartphone and the remote server. The remote stub receives the stream, descrializes it, and invokes the corresponding method. Upon completion, the control is transferred from the remote stub back to the mobile stub by reversing the procedure above. The smartphone then continues the execution, giving the user an illusion that the method call was carried out on the smartphone itself.

Exceptions can arise during the offloading of a remotable method. The mobile stub may receive either no response or an error response from the remote stub, *e.g.*, when a local event is accidentally triggered on the remote server(detected at runtime on remote serverusing similar logging techniques). To handle such exceptions, the mobile stub simply executes the method on the phone itself as if the method was not offloaded. This only causes some extra penalty in execution time and energy.

## 6 Profiling results

We now present the profiling results of several mobile applications on a smartphone. We focus on identifying the remotable methods (RMethods), building the regression functions for each RMethod, and quantifying the profiling overheads. We defer the evaluation of using XRay for offloading applications to remote server at runtime till the next section.

## 6.1 Applications

We summarize the set of applications used in our experiments as follows. All of them are written in C# .NET CE framework to run on WM6.

**Search Indexer** (SI): builds a search index for a set of input files in a directory. The search indexer maintains a hash table containing tokens as keys and document IDs as values. It breaks an input file into tokens delimited by 'space' and inserts them into the hash table.

**Skin Detector (SD):** finds skin colored regions in an input image and reports its percentage. It acts as a content filter for pictures taken by the phone camera and places the new pictures in appropriate directories.

**BarCode reader (BC):** takes an input bitmap image and computes whether the image contains a barcode. If yes, it decodes the barcode in the image.

**PDF2txt (PD):** converts an input PDF file into a text file using the iTextSharp library [28]. The application first removes the images and non-text items, and then converts the raw bytes to text format.

**Chess:** is an interactive, multi-threaded game. It first checks if a user's move is valid and then computes MachineMove. The thinking depth of the machine's algorithm is kept at minimum to attain reasonable response times on a smartphone (half a minute to few minutes).

**MobileAV** (MAV): computes an MD5 checksum of each file in an input directory and matches it against a database of known malware checksums.

#### 6.2 Experimental setup

Table 1 summarizes the hardware specifications of the smartphone and remote server used in our experiments. We burnt a customized kernel image onto the HTC Tytn II phone following the process described in §5. We use the Monsoon power meter [29] to measure the energy consumption on the phone (§7.4).

**Table 1.** *Hardware specifications* 

Device	Specifications						
Mobile	HTC Tytn II, Processor: Qualcomm						
phone	MSM7200TM 400MHz, OS: Windows						
	Mobile 6.1 (CE 5.2), ROM: 256MB, RAM:						
	128MB SDRAM, External SD card: 8GB,						
	Network: 802.11b & 3G						
Remote	Processor: Intel Core 2 Quad Q9300 2.5GHz,						
server	OS: Windows 7, RAM: 8GB PC3-8500 DDR3						
	SDRAM						

To profile each application, we use a set of inputs of different sizes covering the entire range of possible sizes (obtained trivially by either through internet in case of BC, SD, or obtained locally in case of PD, SI). We roughly classify the input sizes into small, medium, and large based on whether they are smaller than 10 KB, between 10 KB and 50 KB, or larger than 50 KB. The phone supports both 3G and WiFi. We control the available WiFi network bandwidth between the phone and remote server by injecting background UDP traffic of different rates (§7.2). We term "Low WiFi bw" as 500 Kbps and "High WiFi bw" as 5000 Kbps. Our 3G bandwidth is 720/1800 Kbps for uplink/downlink.

#### 6.3 Overall results

Table 2 shows the overall profiling and offloading results for the six applications studied in our experiments. We defer the more detailed evaluations on the accuracy of offloading decisions in §7. These applications comprise between 10 to 40 (or over 300 if counting external DLL calls) methods each. After profiling, XRay identifies between 3 to 18 RMethods in five applications. None of the methods in MobileAV or the external DLL calls in PDF2Txt are remotable.

Based on the application profile produced from the profiling phase, XRay generates the partitioned server and mobile code for each application. We then run the partitioned code for our offloading experiments under a variety of scenarios which have varying input sizes and network bandwidth, and measure the number of RMethods that are actually offloaded under each scenario.

The last three columns show the outcome under three example scenarios, which have incrementally higher bandwidth (low WiFi, 3G and high WiFi bandwidth) and larger input size (from small to large). We observe that the offloading logic generated by XRay dynamically makes decisions according to the predicted offloading cost and benefit under different scenarios. For PDF2Txt, it does not offload any RMethods when the bandwidth and input PDF size are small (case 1). As the bandwidth and input size grow (case 2 & 3), it offloads 3 out of the 3 RMethods, leading to shorter total execution times (40% and 83% reduction). For Search Indexer, the 6 RMethods are offloaded 0, 12, and 18 times respectively under the three scenarios, resulting in 39% and 77% reduction in execution times.

In contrast, for Chess, 14 out of the 18 RMethods are always offloaded for each of the 20 machine moves, and hence a total of 280 method invocations are executed on the remote server. This is because the machine thinking time on the phone always dominates the time to offload the fixed-sized chessboard (14KB). Similarly, the 7 RMethods in Barcode Reader are offloaded under all the three scenarios.

**Table 2.** Overall profiling and offloading results for six applications.

Application	Input	Num. of	Num. of	# offloaded RMethods (# times RMethods are offloaded)			
		Methods	RMethods	$\{\% \text{ reduction in execution time}\}$			
				Case 1: Low Wifi	Case 2: 3G & med.	Case 3: High WiFi	
				bw & small input	input	bw & large input	
Search	3 text	18	6	0 (0) {0}	6 (12) {39}	6 (18) {77}	
indexer	files						
Skin detec-	Image	24	10	0 (0) {0}	0 (0) {0}	10 (10) {63}	
tion							
Barcode	Image	12	8	7 (7) {36}	7 (7) {52}	7 (7) {88.5}	
reader							
PDF2Txt	PDF file	14+~300	3 + 0 (exter-	0 (0) {0}	3 (3) {40}	3 (3) {83}	
		(external	nal DLL)				
		DLL)					
Chess	20 human	40	18	14 (280) {81}	14 (280) {84}	14 (280) {89}	
	moves						
MobileAV	1K files	10	0	0 (0) {0}	0 (0) {0}	0 (0) {0}	

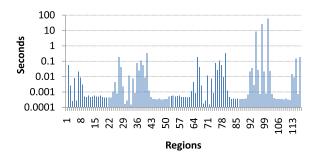


Figure 4. Regions in the Search Indexer event trace

#### **6.3.1** Event trace details

To gain insight into the results in Table 2, we illustrate the system call event traces of Search Indexer and MobileAV captured by XRay on the phone. Figure 4 plots the event trace of Search Indexer. The x-axis shows the sequence of regions where a region is defined as the application execution period between two consecutive local events. The y-axis is the duration of a region in seconds. Out of a total number of 116 regions during the entire execution, 3 regions clearly stand out, lasting from 8.7 to 60.4 seconds, each of which corresponds to one CPU and memory intensive period of inserting the tokens of one input file into the hash table. The 6 RMethods (in Table 2) are executed within these regions, and hence can potentially benefit from offloading given sufficient bandwidth. The event trace of Barcode Reader, PDF2Txt, Skin Detection, and Chess shows similar pattern.

Figure 5 plots the event trace of MobileAV. While there are 33K regions in total, all of them are smaller than 400 milliseconds. This reflects the frequent file read (local) events triggered during the execution of MobileAV. In fact, XRay cannot find any RMethods in MobileAV because all of its methods involve some I/O or GUI activities. Infact, an XRay with replicated file system, which treats file system calls as offloadable, recognizes 6 methods (which consumes more than 99% of runtime) in MobileAV as remotable since 4 remaining functions dealt with GUI.

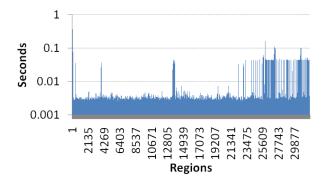


Figure 5. Regions in the MobileAV event trace

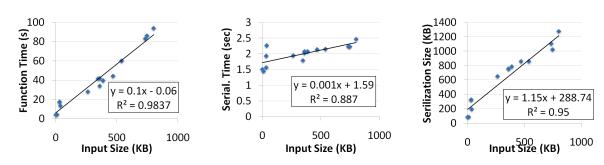


Figure 6. Linear regression for Skin Detection - execution time, serialization time, and serialization size

## 6.4 Regression results

We next show the capability of XRay in automatically generating regression models that capture the dependencies of mobile execution time  $(T_b)$ , serialization size (ssize), and serialization time  $(T_1)$  of an RMethod on input parameter size. As described in  $\S4.2$ , XRay profiles the execution of an application multiple times, each with a different input size, and uses linear regression to derive the models. Figure 6 shows the models generated by XRay for an RMethod in Skin Detection after 20 profiling runs. The  $R^2$  value for the models suggests that they can capture the dependencies fairly accurately.

Figure 7 shows the mobile execution time model in a balloon graph generated by XRay for an RMethod in Search Indexer. It depends on two input parameters, the hash table size and the input file size. We observe that

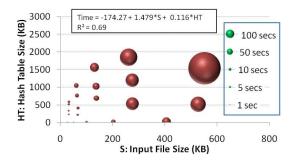
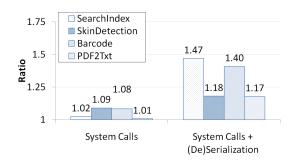


Figure 7. Linear regression for Search Indexer



**Figure 8.** Overhead of event tracing, (de)serialization

despite that both parameters affect the execution time on the phone, the execution time can be captured by a two-variable linear function fairly accurately, as indicated by the  $R^2$  value.

#### 6.5 Profiling overhead

XRay may need to profile an application many times to construct regression models for each RMethod. We study to what extent the event logging and instrumentation of adding serialization and deserialization to each RMethod inflate the application execution time. Figure 8 compares the execution time for four applications — BarCode Reader, PDF2Txt, Search Indexer, and Skin Detection, without logging, with logging, and with both logging and instrumentation. The y-axis is the ratio of the execution time relative to the execution time without logging. It shows that event logging inflates the execution time by smaller than 9% for all the four applications. Even with both logging and instrumentation, the inflation ratio is always under 50%. Such overheads are reasonable since the profiling is conducted offline.

## 7 Offloading results

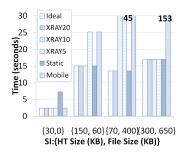
In this section, we evaluate the accuracy of the offloading logic generated by XRay and embedded in the partitioned mobile code during online executions of the applications with apriori unknown input sizes and network bandwidth. We compare the execution time of the following six alternative schemes when running each application on smartphone:

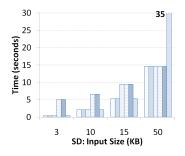
**Mobile:** this is simply to run the original application on smartphone, without any profiling or offloading;

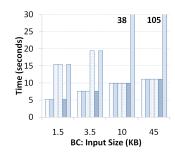
**Static:** this is to use the offloading logic generated by XRay after only one profiling run using a median input size (can be thought as XRay-1); We use this scheme to show the prediction accuracy without linear regression, as done in the previous work [8].

XRay-5/10/20: these three schemes use the offloading logic generated by XRay after 5, 10, and 20 profiling runs respectively; We use these schemes to quantify the effect of the number of profiling runs.

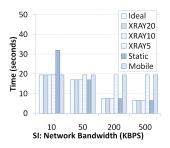
**Ideal:** this is the ideal, hand annotated offloading logic which always makes the correct decisions for any given experimental setting (input size and network bandwidth) during online execution. This scheme is derived by first performing an offline profiling run of the application under the same experimental setting to learn the actual offloading cost and benefit (e.g.,  $T_b$ ,  $T_1$ , ssize, etc.) for each RMethod, and then using this information to make the correct decisions under that experimental setting during online execution.

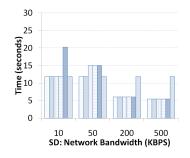


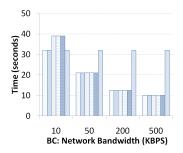




**Figure 9.** Comparison of six offloading schemes for SI, SD, and BC under varying input sizes.







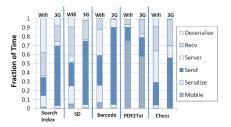
**Figure 10.** Comparison of six offloading schemes for SI, SD, and BC under varying network bandwidth.

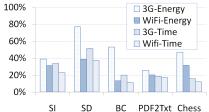
#### 7.1 Offloading decision accuracy under varying input

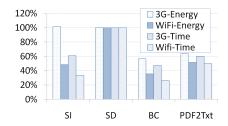
Figure 9 shows the offloading results for Barcode Reader, Skin Detection, and Search Indexer, under four different input sizes, none of which were used in the offline profiling, while fixing the network bandwidth at 3000 Kbps. We make the following observations. First, by comparing the execution time of the Ideal and Mobile schemes, we see that all three application can benefit significantly from offloading. In particular, offloading will benefit Barcode Reader under all four input sizes, but benefit the latter two applications only under the larger input sizes. Second, while the static scheme which uses the offloading logic generated from one profiling run indeed makes the right decisions for Barcode Reader under all four input sizes, it cannot make the right decisions for three out of the four input sizes for Skin Detection, and one out of the four input sizes for Search Indexer. Same is the case with XRay<sub>5</sub> and XRay<sub>10</sub>, which make the right offloading decisions for some input sizes but not for others. Third, XRay<sub>20</sub> which uses the offloading logic generated from 20 profiling runs, always makes the correct offloading decisions and achieves the same running time as the Ideal scheme, for all three applications under all four input sizes. Due to space constraint, we omit the results for Chess and PDF2Txt. PDF2Txt exhibit similar patterns, whereas Chess application has constant input size (size of chess board).

## 7.2 Offloading decision accuracy under varying network bandwidth

Figure 10 shows the offloading results for Barcode Reader, Skin Detection, and Search Indexer, under four different network bandwidth settings, while fixing the input size at 10 KB, 90 KB, and 55 KB, 0 KB respectively. By comparing the execution time of the Ideal and Mobile schemes, we see that the offloading benefits grow with the available network bandwidth for the three applications. In particular, offloading will benefit Barcode Reader when the bandwidth is 50 Kbps or higher, but will not significantly benefit the latter two applications unless the bandwidth is 200 Kbps or higher. The static, XRay<sub>5</sub> and XRay<sub>10</sub> schemes can make the correct offloading decisions only for some of the bandwidth settings but not for others. Finally, XRay<sub>20</sub> always makes the correct offloading decisions and achieves the same running time as the Ideal scheme, for all three applications under all







**Figure 11.** Offloading time breakdown under 3G & WiFi.

Figure 12. Energy consumption under large-sized inputs

**Figure 13.** Energy consumption under medium-sized inputs

four bandwidth settings. The results for Chess and PDF2Txt are similar and are omitted for brevity.

## 7.3 Breakdown of offloading time

To gain insight into the performance bottleneck of the offloading process, we plot for each of the five applications the breakdown of the total offloading time into: (1) mobile time which is the time consumed to execute the stub code on smartphone; (2) serialization and deserialization time on smartphone; (3) time to send and receive the state information across the network; and (4) server time which includes the serialization, deserialization, and execution time of the RMethods on the remote server. The five applications were run under the largest input sizes in Figure 9. Figure 11 shows the time breakdown when the offloading is performed over WiFi and 3G. The average bandwidth is 3000 Kbps under WiFi, and 720 and 1800 Kbps for 3G uplink and downlink.

For all applications except PDF2Txt, the percentage of mobile time is almost negligible, smaller than 5% in all cases. The percentage of mobile time of PDF2Txt is 74% and 55% under WiFi and 3G respectively, because PDF2Txt runs a local method that converts the input PDF file into raw PDF bytes. The method cannot be offloaded since it makes local I/O calls to read the input PDF file. The percentage of server time is between 3 to 35% under WiFi and 3 to 15% under 3G for the five applications. Note that these are the percentages of the total offloading time — even the 35% of offloading time for Chess spent on the remote server under WiFi is a tiny fraction (4%) of execution time of corresponding RMethods on smartphone (if without offloading).

Under WiFi, the applications usually spend more time on sending and receiving state information than on serialization and deserialization. Under 3G, the sending/receiving time clearly dominate the total offloading time, at between 85 to 93% for the applications except PDF2Txt. The percentage of sending time is often far greater than that of receiving time because of the significant difference between the 3G uplink and downlink bandwidth.

#### 7.4 Energy savings

Finally, we study the savings in energy consumption on smartphone as a result of offloading. The energy consumption during an application execution is measured using the Monsoon power monitor [29]. We measure the energy consumption for the five applications, when running Search Indexer, Skin Detection, Barcode Reader, and PDF2Txt with input sizes {135KB, 150KB}, 100KB, 40KB, and 15KB respectively. Search indexer needs two input parameters — the hashtable size and file size. For Chess, we played 20 human moves with the machine.

In Figure 12, the y-axis shows the execution time and energy consumption of offloading the applications under WiFi and 3G relative to those of running the applications entirely on the smartphone. We see that offloading reduces not only the execution time but also the energy consumption. Under 3G, it saves the energy consumption

by between 26% for Skin Detection to 77% for PDF2Txt. The energy savings are even larger under WiFi. This demonstrates the dual benefits of offloading resource-intensive applications: reduced execution time enhances the user experience and reduced energy consumption prolongs the smartphone battery life.

Figure 13 shows similar results for the four applications, using medium input sizes. For Barcode Reader and PDF2Txt, offloading again reduces the execution time and energy consumption simultaneously. For Search Indexer, however, the energy consumption of offloading under 3G increases just a little compared to that of not offloading, in spite of the 39% decrease in execution time. This example highlights the importance of considering both energy saving and performance improvement in making offloading decisions, which we leave as future work.

#### 8 Conclusion

We presented XRay, a tool that automatically identifies remotable methods for smartphone applications and further determines whether offloading the methods will benefit the applications. XRay works by tracing a selected set of system call events issued during application execution, and identifies remotable methods as those that do not contain any local events (*e.g.*, GUI, I/O, and network). For each remotable method, XRay further accurately estimates the key components that affect offloading time, including suspending/resuming application execution and the size of the state that needs to be transferred over the network. We have implemented XRay on WM6 and applied it to a set of smartphone applications, *e.g.*, barcode reader, search indexer, and document converter. Our experiments confirm that with a small number of profiling runs, XRay can automatically generate offloading decision logic for each remotable method that makes correct offloading decisions in future online executions of these applications under varying input parameters and network conditions. We believe XRay makes a significant contribution towards enabling thousands of existing mobile applications to benefit from cloud computing resources. Further, it can be used to assist application developers in developing future "cloud-aware" mobile apps.

#### References

- [1] "Apple iphone app store," http://www.apple.com/iphone/apps-for-iphone/.
- [2] B.-G. Chun and P. Maniatis, "Augmented Smartphone Applications Through Clone Cloud Execution," in Proc. of HotOs, 2009.
- [3] M. Satyanarayanan, P. Bahl, R. Cceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.
- [4] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. i Yang, "The case for cyber foraging," in 10th ACM SIGOPS, 2002.
- [5] R. Balan, M. Satyanarayanan, S. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," 2003. [Online]. Available: citeseer.ist.psu.edu/balan03tacticsbased.html
- [6] J. Flinn, D. Narayanan, and M. Satyanarayanan, "Self-tuned remote execution for pervasive computing," in Proc. of HOTOS, 2001.
- [7] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Proc. of ICDCS*, 2002.
- [8] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proc. of MobiSys*, 2010.
- [9] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based Partitioning for Sensornet Applications," in *Proc. of NSDI*, 2009.
- [10] G. C. Hunt and M. L. Scott, "The coign automatic distributed partitioning system," in OSDI, 1999.
- [11] J. F. Shafeeq, J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan, "Data staging on untrusted surrogates," in *Proc. of FAST*, 2002.
- [12] S. Goyal and J. Carter, "A lightweight secure cyber foraging infrastructure for resource-constrained devices," in *Proc. of WMCSA*, 2004.
- [13] Y.-Y. Su and J. Flinn, "Slingshot: deploying stateful services in wireless hotspots," in *Proc. of Mobisys*, 2005.
- [14] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proc. of OSDI*, 2004.

- [15] "Event tracing for windows (etw)," http://msdn.microsoft.com/en-us/library/ms751538.aspx.
- [16] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma, "Automated known problem diagnosis with event traces," in *EuroSys*, 2006.
- [17] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems," in OSDI, 2008.
- [18] M A Effroymson, "Multiple regression analysis," in Ralston, A., and Wilf, H. S. (Eds.), Mathematical Methods for Digital Computers, Wiley, New York, 1960.
- [19] "Handwritten c# parser," http://csparser.codeplex.com/.
- [20] "The r-project for statistical computing," http://www.r-project.org.
- [21] "Strace," http://linux.die.net/man/1/strace.
- [22] "Celog event tracking," http://msdn.microsoft.com/en-us/library/aa462467.aspx.
- [23] Y. Liu, A. Rahmati, Y. Huang, H. Jang, L. Zhong, Y. Zhang, and S. Zhang, "xshare: supporting impromptu sharing of mobile phones," in *Proc. of Mobisys*, 2009.
- [24] M. Becher and R. Hund, "Kernel-level interception and applications on mobile devices," in *Technical Report, University of Mannheim*; *TR-2008-003*, 2008.
- [25] "Setkmode," http://msdn.microsoft.com/en-us/library/aa908769.aspx.
- [26] "Microsoft platform builder," http://msdn.microsoft.com/en-us/library/ms938344.aspx.
- [27] "Windows embedded ce shared source," http://msdn.microsoft.com/en-us/windowsembedded/ce/dd567722.aspx.
- [28] "itextsharp," http://itextsharp.sourceforge.net/.
- [29] "Monsoon power monitor," http://www.msoon.com/ LabEquipment/PowerMonitor/.