Improving the Performance and Reliability of Mobile Applications

by

Lenin Ravindranath Sivalingam

S.M., Computer Science, Massachusetts Institute of Technology (2010)
B.E., Computer Science, Anna University (2006)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Signature redacted

Author .........

Department of Electrical Engineering and Computer Science

August 26, 2014

Signature redacted

Certified by ...................................................

Hari Balakrishnan
Professor
Thesis Supervisor

Signature redacted

Accepted by ...........................................

Vesela A. Kolodziejski
Chairman, Department Committee on Graduate Theses
Abstract

The mobile application ("app") ecosystem has grown at a tremendous pace with millions of apps and hundreds of thousands of app developers. Mobile apps run across a wide range of network, hardware, location, and usage conditions that are hard for developers to emulate or even anticipate during lab testing. Hence, app failures and performance problems are common in the wild. Scarce resources, shift away from familiar synchronous programming models, and poor development support has made it more difficult for app developers to overcome these problems. This dissertation focuses on systems that make it significantly easy for app developers to diagnose and improve their mobile apps.

To reduce user annoyance and survive the brutally competitive mobile app marketplace, developers need systems that (i) identify potential failures before the app is released, (ii) diagnose problems after the app is deployed in the wild, and (iii) provide reliable app performance in the face of varying conditions in the wild. This dissertation presents systems that satisfy these needs. VanarSena makes it easy to diagnose common failures in mobile apps before deployment, AppInsight makes it easy to monitor mobile apps after deployment, and Timecard allows mobile apps to adapt to conditions in the wild and provide consistent performance. For the legion of amateur app developers with fewer resources at hand, these systems significantly reduce the barrier for diagnosing and improving mobile apps.

The systems are built on top of a binary instrumentation framework that automatically rewrites app binary at bytecode level. Hence, using them requires minimal effort on part of the app developer. The systems include novel instrumentation techniques to automatically track the runtime behavior of the app. To cope with the scarcity of resources, they include resource-aware mechanisms that incur negligible overhead. To make them immediately deployable, they are designed to require no modification to the OS or runtime.

We have built VanarSena, AppInsight, and Timecard for the Windows Phone platform. VanarSena does automated app testing by systematically emulating user interactions and fault conditions from the wild to uncover app failures. VanarSena uncovered 2,969 distinct crashes in more than 1,100 apps in the app store. AppInsight does light-weight monitoring of mobile apps in the wild. It automatically instruments the app binary to track performance and failures. AppInsight uncovered several performance bottlenecks and crashes in the wild and has provided useful feedback to developers. Timecard enables apps to adapt at runtime and provide consistent performance in the face of varying conditions in the wild. Timecard can tightly control the response time around a desired user-perceived delay.
Acknowledgments

I would like to thank my advisor, Hari Balakrishnan, for his constant guidance and advice on my projects. Over the past six years, Hari has been a friend, a mentor, and a great source of support. Hari’s passion for research, his ability to generate and position ideas, and his writing skills are truly inspiring. I am indebted to him for the patience he showed during the times I was away from MIT. I would also like to express my gratitude to Sam Madden, who has been like a co-advisor to me at MIT. After Hari, Sam has been my next stop for advice on research.

Jitu Padhye has been an incredible mentor and collaborator for the past several years. Without his sustained encouragement and enthusiasm, this thesis would not have been possible. Besides being a caring research mentor, Jitu has also been a pillar of support on matters beyond research. Victor Bahl has been a great mentor and a guide ever since my first internship as an undergraduate in his group. His encouragement motivated me to go to graduate school, and his incredible support helped me throughout my Ph.D.

Ratul Mahajan has been an amazing collaborator and mentor. His ideas and insights laid solid foundations to this thesis. I have also benefited immensely from my collaboration with Suman Nath, and I want to thank him for his valuable contributions to this thesis. I also want to thank Sharad Agarwal, Ian Obermiller, Shahin Shayandeh, and Ronnie Chaiken for their contributions to AppInsight.

Arvind Thiagarajan has been an awesome collaborator, office-mate, and a well-wisher. I thoroughly enjoyed brainstorming, hacking, and building systems with him, and I want to thank him for those great times at MIT. I thank Alec Wolman, my first internship mentor, for introducing me to research. I thank Ranveer Chandra for his constant mentorship. I always looked up to him in defining my style of research. I want to thank Yuan Yu for the exciting summer at Microsoft Research Silicon Valley. I am grateful to Chandu Thekkath, Venkat Padmanabhan, and Ram Ramjee for their guidance when I was still new to research. I want to thank all my amazing collaborators on various projects. I learnt so many things working with each one of them.

Ganesh Ananthanarayanan has been a great friend and my first stop for any help. I want to thank him for all the patience he has showed towards me. I want to thank Ramesh Chandra and Matthai Philipose for their encouragement and positive spirit whenever I reached out to them. Eugene Wu is the most down-to-earth person I have met at MIT and I want to thank him for all the wonderful conversations we had. I thank my office-mates, lab-mates, colleagues, and friends at MIT and MSR, and all my friends at school, college, and elsewhere, for their support.

I am extremely grateful and indebted to my parents, my sister, and my brother-in-law for their love, care, and motivation all through these years. I cannot express enough gratitude and thanks to my wife, Thangam, for her love, affection, patience, and constant support, and for standing by me through the highs and lows of graduate school.
Contents

1 Introduction .......................................................... 15
  1.1 Tracking Mobile App Execution .................................. 18
    1.1.1 Tracking User Transactions .............................. 19
  1.2 VanarSena: Testing Before Deployment ......................... 20
  1.3 AppInsight: Monitoring In The Wild .......................... 21
  1.4 Timecard: Adapting To Variable Conditions .................. 23
  1.5 Contributions and Roadmap ................................... 25

2 Tracking User Transactions ........................................ 27
  2.1 Mobile App Programming Pattern ................................ 27
  2.2 Binary Instrumentation of Mobile Apps ....................... 31
  2.3 Instrumentation to Track User Transactions .................. 31
    2.3.1 Tracking User Interaction .............................. 35
    2.3.2 Tracking thread execution .............................. 35
    2.3.3 Matching async calls to their callbacks ................ 36
    2.3.4 Tracking Thread Synchronization ....................... 39
    2.3.5 Tracking UI updates ................................... 43
    2.3.6 Tracking unhandled exceptions ........................ 43
    2.3.7 Example trace ........................................... 43
  2.4 Supporting Other Platforms .................................. 44

3 Automated App Testing ............................................ 47
  3.1 App Crashes in-the-Wild ...................................... 48
    3.1.1 Data Set .................................................. 49
    3.1.2 Root Causes of Observed Crashes ....................... 49
    3.1.3 Findings .................................................. 50
  3.2 Goals and Non-Goals .......................................... 54
  3.3 Architecture .................................................. 56
  3.4 Instrumentation ............................................... 58
  3.5 UI Automator .................................................. 58
    3.5.1 Hit Testing ............................................... 59
    3.5.2 When to interact next? .................................. 60
    3.5.3 Randomized Concurrent Monkeys ......................... 61
  3.6 Inducing Faults ............................................... 61
  3.7 Evaluation .................................................... 64
    3.7.1 Crashes ................................................... 64
    3.7.2 Monkey Techniques ....................................... 69
### Table of Contents

3.8 Discussion and Limitations ........................................ 74
3.9 Related work ......................................................... 75
3.10 Chapter Summary .................................................... 77

4 Monitoring In The Wild .................................................. 79
4.1 Goals ................................................................. 80
4.2 AppInsight Design Overview ........................................ 80
4.3 Instrumenter .......................................................... 82
  4.3.1 Capturing Additional Information ............................ 82
4.4 Analysis Methodology ............................................... 82
  4.4.1 User transactions .............................................. 82
  4.4.2 Critical Path ................................................... 83
  4.4.3 Exception path ............................................... 85
  4.4.4 Aggregate Analysis .......................................... 85
4.5 Developer Feedback ................................................. 85
4.6 Results .............................................................. 86
  4.6.1 Deployment ..................................................... 86
  4.6.2 User Transactions and Critical Paths ....................... 89
  4.6.3 Exception paths ............................................... 90
  4.6.4 Aggregate Analysis .......................................... 90
  4.6.5 Case Studies .................................................. 92
  4.6.6 Micro-benchmarks ............................................ 93
4.7 Discussion and Limitations ......................................... 95
4.8 Related Work ........................................................ 95
4.9 Chapter Summary .................................................... 96

5 Adapting To Variable Conditions ...................................... 99
5.1 Timecard Architecture ............................................. 100
5.2 Tracking elapsed time .............................................. 102
  5.2.1 Transaction tracking ......................................... 102
  5.2.2 Synchronizing time .......................................... 104
5.3 Predicting Remaining Time ......................................... 107
  5.3.1 Predicting $N_2$ ............................................. 107
  5.3.2 Predicting $C_2$ ............................................. 108
5.4 Implementation ...................................................... 109
5.5 Evaluation .......................................................... 109
  5.5.1 Is Timecard Useful? .......................................... 110
  5.5.2 End-to-End Evaluation .................................... 113
  5.5.3 Prediction Accuracy ......................................... 116
  5.5.4 TimeSync .................................................... 119
  5.5.5 Overhead ..................................................... 120
5.6 Discussion and Limitations ......................................... 121
5.7 Related Work ........................................................ 122
5.8 Chapter Summary .................................................... 122

6 Conclusion ............................................................ 125
6.1 Summary ............................................................ 125
6.2 Future Work ........................................................ 126
List of Figures

1-1 Systems presented in this dissertation. ............................................. 17
1-2 Example of asynchronous coding pattern. ............................................ 18
1-3 Execution trace for the code in Figure 1-2 ........................................ 19
1-4 Execution trace of an app crash. ....................................................... 20
1-5 A typical transaction in a cloud-based mobile app. The processing at the app
and the server can span multiple threads and asynchronous calls. Timecard
provides two API calls to query at the server: 1) estimates the elapsed time
since the initiation of the user request and 2) predicts the remaining time
for the mobile app to receive and render the response. The server can use
these estimates to adapt its work time to tightly control the end-to-end, user-
perceived delay. ................................................................. 23

2-1 User interface for a sample mobile app. When the user clicks on the search
button, the app fetches data from Twitter, processes the data, and updates
the UI with the search result. .................................................. 28
2-2 Asynchronous coding pattern in mobile apps. ...................................... 28
2-3 Execution trace for the code in Figure 2-2. ....................................... 29
2-4 CDF of number of edges in the execution graph for 167,000 tasks from 30
popular Windows Phone apps used by 30 users for over 6 months. There are
execution graphs with more than 8000 edges. ...................................... 29
2-5 Execution trace of a location-based app. ............................................ 30
2-6 Execution trace of an app crash. ....................................................... 31
2-7 MSIL code for the btnFetch_Click method in Figure 2-2. ......................... 32
2-8 Instrumentation of MSIL bytecode. The start and end of the btnFetch_Click
method is instrumented to call external tracker methods. A static identifier
(5) is passed to the tracker methods. ............................................. 33
2-9 Structure of Instrumented App. ....................................................... 34
2-10 Instrumented version of the code in Figure 2-2. The actual instrumentation
is done on MSIL byte code. We show decompiled C# code for convenience. . 37
2-11 Detour library .......... ............................................................. 38
2-12 Delayed callback ....... ............................................................ 39
2-13 Instrumentation for delayed callbacks. ............................................. 40
2-14 Capturing User Transactions. ......................................................... 40
2-15 Thread synchronization and causal edges. ......................................... 41
2-16 Thread synchronization example where the main thread waits for two pro-
cessing threads to complete. ....................................................... 41
2-17 Instrumented code for the thread synchronization example in Figure 2-16. .. 42
2-18 Tracking thread synchronization edges for the example in Figure 2-16. .... 43
3-1 CDF of crash reports per app. ................................................. 49
3-2 Stack trace fragment for Chicago Transit Tracker crash. The exception was
WebException. ................................................................. 49
3-3 Cumulative distribution of bucket sizes ..................................... 50
3-4 Distinct stack traces in various buckets for one particular app ........ 51
3-5 Distinct stack traces in various buckets for all apps ..................... 51
3-6 Example app pages. UI elements pointed by red arrows can be interacted
with. Arrows and numbers will be explained in Section 3.5.1. ............ 55
3-7 App structure for the example in Figure 3-6. ............................. 56
3-8 VanarSena Architecture. .................................................... 57
3-9 Monkey design. ............................................................... 57
3-10 Event Handlers are instrumented to enable Hit Testing. Handler's unique id
is 12. ............................................................................. 59
3-11 Generating ProcessingCompleted event. .................................... 60
3-12 UI automator flow. .................................................................. 62
3-13 Intercepting web API to proxy through web response FIM and informing
network FIM about the impending network transfer. ....................... 62
3-14 Crashes per app .................................................................... 65
3-15 Coverage of crash buckets in WPER data ................................. 67
3-16 FIMs causing crashes ............................................................ 68
3-17 Fraction of invokable to total controls and unique event handlers to total
controls in an app. ..................................................................... 70
3-18 Time to run apps with and without hit testing ............................ 70
3-19 Fraction of pages covered with and without hit testing. ............... 71
3-20 Fraction of transactions covered with and without hit testing .......... 71
3-21 Processing times for transaction. .............................................. 72
3-22 Fraction of pages covered by runs compared to pages covered by 10 runs. 73
3-23 Fraction of transactions covered by runs compared to transactions covered by
10 runs. ............................................................................. 73

4-1 AppInsight System Overview ..................................................... 81
4-2 Transaction Graph for the trace in Table 2.1. ................................. 83
4-3 Transaction Graph for transaction in Figure 2-5. Record labels are omitted
for simplicity. ....................................................................... 84
4-4 Collage of some of the views in the developer feedback UI. .......... 86
4-5 CDF of user-transaction duration. .............................................. 87
4-6 CDF of number of edges in user transactions and in critical paths. The X-axis
has been clipped at 100. The top line ends at 347, and the bottom ends at
8,273. ............................................................................. 89
4-7 Cumulative fraction of time in the critical path as a function of number of
edges. ............................................................................. 90
4-8 Variation in transaction length for each group. Both axes are clipped at 20. 91

5-1 A typical user transaction in a server-based mobile app. .................. 101
5-2 Timecard Architecture. ........................................................... 102
5-3 A location-based app that queries a server. ................................. 103
RTTs of probes from an app to a server with Timecard, when the network is busy, and when the radio is either idle or busy. (Note: There is no high-power idle state in LTE.) Timecard's probe transmissions strategy results in lower variability.  

$C_2$ from one of the data-intensive user transactions in the Facebook app.  

Size of data downloaded by apps in the AppInsight and PhoneMonkey datasets.  

Mean user-perceived response time and its standard deviation for different transactions.  

Client processing delays.  

Network transfer delays.  

User-perceived delays for two apps. With Timecard, delays are tightly controlled around the desired value.  

Elapsed time $(C_1 + N_1)$ from two apps.  

Deadline control errors: discrete work time in one app and $N_2$ prediction errors.  

Trade-off between server work time and quality of results.  

Accuracy of $N_2$ and $C_2$ prediction.  

Clock drift in smartphones.
List of Tables

2.1 Trace of code in Fig. 2-10. The UI thread id is 0. Method id and call id are shown within parenthesis. .................................................. 44

3.1 Examples of crash buckets and corresponding root causes, categories, and ways to induce the crashes ........................................ 53
3.2 Number of crashed apps for various ratings ........................................ 65
3.3 Crashes found by VanarSena. .................................................. 67

4.1 Summary statistics from our deployment .................................................. 87
4.2 Summary statistics for 13 of the 30 apps. For conciseness, we highlight a single app out of each of the major app categories. The name of the app is anonymized. Overhead data is explained in §4.6.6. ........................................ 88
4.3 Overhead of AppInsight Logger. Averaged over 1 million runs, on a commonly used phone model. .................................................. 94

5.1 Transaction context. The three timestamps are named as per Figure 5-1. 103
5.2 Data sets used for evaluation of Timecard. .................................................. 110
Previously Published Material

This dissertation includes material from the following publications:


Chapter 1

Introduction

This dissertation is concerned with improving the performance and reliability of mobile applications.

The mobile application (“app”) ecosystem has grown at a tremendous pace with millions of apps and hundreds of thousands of app developers. Users rely on these apps for a variety of tasks, from checking weather and news to banking and shopping. For mobile apps, improving performance and reliability is less about making sure that “mission critical” software is bug-free, but more about survival in a brutally competitive marketplace. Because the success of an app hinges on good user reviews, even a handful of poor reviews can doom an app to obscurity. A scan of reviews on mobile app stores shows that an app that crashes or has poor performance is likely to garner negative reviews.

Unlike traditional “enterprise” software, mobile apps are typically used in more uncontrolled conditions, in a variety of different locations, over different wireless networks, with a wide range of input data from user interactions and sensors, and on a variety of hardware platforms. Coping with these issues is difficult enough for sophisticated developers and well-funded organizations, but for the legion of less-experienced developers with fewer resources at hand, the problem is acute. Hence, app crashes and performance problems are common in the wild.

This dissertation presents three closely related systems that make it easy for mobile app developers to diagnose and improve the performance and reliability of their apps. These systems operate at different stages in the app development and deployment pipeline (Figure 1-1). VanarSena is a system that makes it easy for developers to test their apps and uncover app failures before deployment. AppInsight makes it easy for developers to monitor app performance and failures in the wild after deployment. Timecard makes it easy for mobile apps to adapt at runtime and provide consistent performance in the face of varying conditions in the wild.

Testing: To provide acceptable user experience, it is important to uncover and fix app failures before the app is released to the users. But today, most developers lack the resources and time to thoroughly test their apps under different environmental and usage conditions. To help developers uncover app failures before deployment, we built VanarSena. VanarSena is an easy to use, automated, and scalable system that thoroughly tests mobile apps for common crashes that occur in the wild. In VanarSena, the developer simply submits the app to a cloud service, and then within a short amount of time obtain a detailed report of app crashes. Developers can fix these failures before submitting to the app store.
VanarSena does automated dynamic testing by systematically emulating user interactions and fault conditions from the wild. Automatically testing mobile apps is challenging because of their UI-centric and asynchronous nature (§1.1). VanarSena overcomes these challenges by adopting a gray box testing method, instrumenting the app binary to achieve both high coverage and high speed. The techniques in VanarSena are driven by a study of 25 million real-world crash reports from more than 100,000 Windows Phone apps reported in 2012. VanarSena runs several app “monkeys” in parallel to emulate user, network, and sensor data behavior, returning a detailed report of crashes for common root causes.

**Monitoring:** VanarSena helps to significantly reduce common crashes that occur in the wild, but does not find all possible crashes. Given that we execute apps in an emulated setting, VanarSena cannot accurately characterize the performance of the app in the wild and hence find performance problems. Lab testing is important, but is not sufficient. The mobile environment is complex and varying, and a full range of usage conditions is difficult to emulate in a test setting. Thus, in addition to testing, developers must continue to monitor how the apps perform in the wild after it is deployed. To improve the quality of apps, collection of diagnostic and trace data from the field is essential.

Today, there is little platform support for app monitoring in the field. Major mobile platforms, including iOS, Android, and Windows Phone, report crash logs to developers, but it is often difficult to identify the causes of crashes from these logs [5], and this data does not help diagnose performance problems. Thus, the only option left is for the developer to include custom tracing code the app. However, as we will show in Section 1.1, writing such code is no easy task given the highly asynchronous programming pattern used in apps (§1.1). This challenging task is made even more difficult because tracing overhead must be minimized to avoid impact on app performance, and also to limit the consumption of scarce resources such as battery and network bandwidth.

We built AppInsight, which addresses the above challenges and makes it easy for app developers to understand how their app is performing in the hands of users. AppInsight automatically instruments the app to collect performance and failure data from the wild. It carefully selects code points to instrument to minimize overhead. AppInsight analyzes the collected data and provides detailed feedback to the developer about performance bottlenecks, failures, and their root causes. Using AppInsight, developers can iteratively improve the performance and reliability of their mobile apps.

**Runtime Adaptation:** In interactive mobile apps, users expect a timely response for their interactions. Responses that arrive within a predictable period of time improve the user experience, whereas the failure to provide consistent response times can have adverse financial implications for even small degradations in response times [43, 18, 77].

AppInsight enables developers to improve response time by identifying bottlenecks and pinpointing the root causes. But, providing consistent response times is still challenging, because there are several variable delays between the start of a user's request and the completion of the response. These delays include location lookup, sensor data acquisition, radio wake-up, network transmissions, processing in the app, etc.

To provide consistent performance, the app needs to adapt at runtime to the variable conditions. For example, when the network is slow, the app can reduce its processing time and trade-off on the quality of result to meet a desired response time. When the network is fast, the app can increase its processing time to produce a better quality result.
We built Timecard to enable apps to adapt and provide consistent performance. In Timecard, we specifically focus on mobile apps that use servers in the cloud for their core functionality since user interactions that involve communication to a server have long and variable delays. Timecard instruments both the app and the server to automatically estimate the variable delay components at runtime. During an user interaction, it exposes these estimates to the communicating server through two simple API calls. Using this API, the server can adapt its core processing to tightly control the end-to-end response delay in the app. The server can trade-off on the quality or quantity of the result to control the delay. Timecard uses several techniques to overcome the challenges in estimating the delay components at runtime.

**Design:** The design of VanarSena, AppInsight, and Timecard was guided by two key design goals. (i) *Minimal developer effort:* The systems should be very easy to use. App developers should write zero to minimal additional lines of code. (ii) *Ready deployability:* The systems should be immediately deployable and quickly updatable. They should not require modifications to the OS or runtime which typically creates a significant barrier for adoption.

To achieve these design goals, we built all three systems on top of a binary instrumentation framework that automatically instruments app binaries at bytecode level. The instrumentation framework can unpack the app binary, inject new bytecode or rewrite existing bytecode, and repack the binary without any developer support.

VanarSena instruments the app binary to dynamically track the app execution state and efficiently test the app for failures. AppInsight instruments the app binary to automatically monitor app performance and failures in a light-weight manner in the wild. Timecard instruments app and server binaries to automatically track the delay of various components at runtime. Using VanarSena and AppInsight requires zero developer effort. We do not require app developers to write additional code, or add code annotations. Timecard provides
void btnFetch_Click(object sender, RoutedEventArgs e) {
    var req = WebRequest.Create(url);
    req.BeginGetResponse(reqCallback, null);
}

void reqCallback(IAsyncResult result) {
    /* Process */
    UIDispatcher.BeginInvoke(updateUI);
}

void updateUI() {
    /* Update UI */
}

Figure 1-2: Example of asynchronous coding pattern.

a simple API and incorporating Timecard into an app or a server requires minimal developer effort. All three systems require no changes to the OS or runtime.

The key component common to these systems is automatic instrumentation to dynamically track app execution, performance, and failures. As we discuss next, this task is challenging because of the highly asynchronous programming patterns in apps.

1.1 Tracking Mobile App Execution

Mobile apps are UI-centric in nature. In modern UI programming frameworks [26, 71], the UI is managed by a single, dedicated thread. All UI updates, and all user interactions with the UI take place on this thread. To maintain UI responsiveness, applications avoid blocking the UI thread as much as possible, and perform most work asynchronously. Some mobile-programming frameworks like Silverlight [71], do not even provide synchronous APIs for time-consuming operations like network I/O and location queries. Even compute tasks are typically carried out by spawning worker threads. Thus, user requests are processed in a highly asynchronous manner.

This is illustrated in Figure 1-3, which shows the execution trace for a simple code snippet in Figure 1-2. In the figure, horizontal line segments indicate time spent in thread execution, while arrows between line segments indicate causal relationships between threads. When the user clicks the button, the OS invokes the event handler (btnFetch_Click) in the context of the UI thread; the handler makes an asynchronous HTTP request; when the HTTP request completes, the OS calls reqCallback in a worker thread which processes the fetched data; when the processing finishes, the worker thread invokes the UI Dispatcher, to queue a UI update to run in the UI thread. The user-perceived delay for this interaction is the time between the user interaction and the UI update. To track the execution of this interaction and measure its performance, we need to instrument the app to track the entire execution graph in Figure 1-3 spanning multiple threads and asynchronous calls. Real apps are much more complex with hundreds of edges in the execution graph.

Failure analysis is also complicated by the asynchronous nature of the app. Consider the example in Figure 1-4. Suppose the app crashes in the method parseURL(), which is called
Background worker thread

Download Delay

Web Request Call

UI Dispatch

UI Thread

UI Event Handler Start

UI Event Handler End

User Interaction

UI Update

User-perceived delay

Figure 1-3: Execution trace for the code in Figure 1-2

in a worker thread that started at parseXML(). Since the UI thread function that started the web request has exited, the OS has no information about the user context for this crash. Thus, in the exception log offered by today's popular mobile platforms, the developer will only see the stack trace of the crashed thread, from parseURL() to parseXML(). The developer however, might want more information, such as the user interaction that triggered the crash, to speed up debugging. This underscores the need for tracking user transactions across thread boundaries.

1.1.1 Tracking User Transactions

We call the execution trace in Figure 1-3, a user transaction since it is triggered by the user and represents a single activity in the app. To monitor app performance and failures, our goal is to instrument the app to track its execution as a set of user transactions. A user transaction begins with a user interaction of the UI, and ends with completion of all synchronous and asynchronous tasks (threads) in the app that were triggered by the interaction. To automatically track user transactions in a light-weight manner, we carefully select code points to instrument in the app. Chapter 2 explains our instrumentation techniques in detail.

VanarSena uses binary instrumentation and the notion of user transactions to efficiently test the app and reason about failures. AppInsight collects data about user transactions from the wild and provides feedback to the developers about critical paths and exception paths in a user transaction. Timecard tracks user transactions at runtime, measures the delays of different components in a transaction, and exposes them to apps through a simple API.

In the following sections, we describe the three systems we have developed and the key insights from each of the systems.
1.2 VanarSena: Testing Before Deployment

VanarSena does automated app testing by systematically emulating user interactions and fault conditions from the wild to uncover app failures. Using VanarSena, developers can identify and fix problems that can potentially occur in the hands of users, before the app is released.

The starting point in the design of VanarSena is to identify what types of faults have the highest “bang for the buck” in terms of causing real-world failures. To this end, we studied 25 million crash reports from more than 100,000 Windows Phone apps reported in 2012. Three key findings inform VanarSena’s design:

1. Over 90% of the crashes were attributable to only 10% of all the root causes observed.
2. Although the “90-10” rule holds, the root causes affect a wide variety of execution paths in an app.
3. A significant fraction of these crashes can be mapped to well-defined externally inducible events, such as invalid text inputs and unhandled HTTP error codes.

The first finding indicates that focusing on a small number of root causes will improve reliability significantly. The second suggests that the fault finder needs to cover as many execution paths as possible. The third indicates that software emulation of user inputs, network behavior, and sensor data is likely to be effective, even without deploying on phone hardware.

Using these insights, VanarSena is built to find common faults in mobile applications. The developer uploads the app binary to our service, along with any supporting information such as a login and password. VanarSena instruments the app, and launches several *monkeys* to run the instrumented version on phone emulators. As the app is running, VanarSena emulates a variety of user, network, and sensor behaviors to uncover and report observed failures.
A noteworthy principle in VanarSena is its “greybox” approach, which instruments the app binary before emulating its execution. Greybox testing combines the benefits of “white-box” testing, which requires detailed knowledge of an app’s semantics to model interactions and inputs, but isn’t generalizable, and “blackbox” testing, which is general but not as efficient in covering execution paths. VanarSena includes the following novel techniques:

- **Hit Testing**: The use of binary instrumentation enables a form of execution-path exploration we call hit testing, which identifies how each user interaction maps to an event handler and in turn a user transaction. Hit testing allows VanarSena to cover many more execution paths in a given amount of time by skipping non-invokable controls and invoking only those controls that lead to unique user transactions.

- **ProcessingCompleted Event**: Binary instrumentation and tracking user transactions allows VanarSena to determine when to emulate the next user interaction in the app. By tracking user transactions at runtime, we generate a ProcessingCompleted signal that indicates that processing of the previous interaction is complete. The use of ProcessingCompleted signal significantly reduces testing time.

- **Fault Inducers**: App instrumentation also makes VanarSena extensible, by inserting our own event handlers that trigger under certain situations, such as network calls and certain user actions. VanarSena can then trap these event handlers to induce specific faults such as emulating slow or faulty networks. We have written several such fault inducers, and more can be easily written.

We have implemented VanarSena for Windows Phone apps, running it as an experimental service. We evaluated VanarSena empirically by testing 3,000 apps from the Windows Phone store for commonly-occurring faults. The key results are summarized below:

- VanarSena discovered failures in 1,108 of these apps, which have presumably undergone some testing and real-world use. Overall, VanarSena uncovered 2,969 distinct crashes, including 1,227 that were not previously reported.

- Hit testing accelerates testing by avoiding interacting with non-invokable controls and interacting with only those that lead to unique user transactions. It speeds-up testing time by 2x on average and up to 20x for some apps without compromising on app coverage.

- Generation and use of ProcessingCompleted event significantly improves testing time. Compared to using a static timeout of 4 seconds which waits for 90% of the interactions to finish, the use of ProcessingCompleted speeds-up testing time by 3x on average and up to 30x for certain apps.

### 1.3 AppInsight: Monitoring In The Wild

AppInsight does light-weight monitoring of mobile apps in the wild. AppInsight automatically instruments the app binary to track performance and failures. In AppInsight, the developer simply passes the app binary through an instrumentation tool which produces the instrumented version of the app. The developer does not need to write any additional code,

---

1. Thus, VanarSena would be even more effective during earlier stages of development
or add code annotations. The instrumented version of the app is submitted to the app store. When users run the instrumented app, trace data is collected and uploaded to a server. The trace data is analyzed and the findings are made available to the developer via a web-based interface.

The key techniques and the feedback provided to the developer in AppInsight is summarized below:

- **User-Perceived Delay**: AppInsight tracks the execution of the app as a set of user transactions. AppInsight correctly stitches the threads and asynchronous components together into a cohesive transaction graph. The duration of the user transaction represents the user-perceived delay and this information helps the developer understand the user experience in the wild.

- **Critical Path**: The critical path is the bottleneck path in a user transaction, such that changing the length of any part of the critical path will change the user-perceived latency. AppInsight provides the developers with information on the critical path through their code for every user transaction. This information points the developer to the optimizations needed for improving user experience.

- **Exception Path**: While critical path is useful for understanding performance bottlenecks, to debug app crashes, AppInsight provides the developer with exception paths. The exception path is the path from the user interaction to the exception method, spanning asynchronous boundaries. This information provides the developer with more context compared to just stack traces. The entire execution trace in Figure 1-4 is the exception path for the crash.

- **Aggregate Analysis**: AppInsight also analyzes transactions in aggregate. By examining transactions from multiple users, AppInsight automatically identifies common critical paths, outliers, and root causes of performance variability. This information further helps the developer focus development effort on components that matter in the wild.

We have implemented AppInsight for the Windows Phone platform. To evaluate AppInsight, we instrumented 30 popular apps and recruited 30 users to use these apps on their personal phones for over 4 months. This deployment yielded trace data for 6,752 app sessions, totaling over 33,000 minutes of usage time. Our evaluation shows that:

- AppInsight is lightweight – on average, it increases the run time by 0.021%, and the worst-case overhead is less than 0.5%. The memory overhead is just 2% and network overhead is only 4%. The instrumentation and battery overhead are negligible.

- Despite the low overhead, the instrumentation is comprehensive enough to allow us to make several detailed observations about app performance and failures in the wild. AppInsight uncovered several performance bottlenecks and crashes and has provided useful feedback to developers. Developers have used the feedback to improve the performance and reliability of their apps.
Timecard provides two simple API calls for servers to query. Given the desired user-perceived delay, Timecard allows the server to obtain answers to the following questions:

1. **GetElapsedTime()**: How much time has elapsed since the initiation of the request at the mobile app?
2. **PredictRemainingTime(bytesInResponse)**: How much time will it take for the mobile app to receive an intended response over the network and then process it?

The server can use the difference between the desired delay bound and the sum of the elapsed time and predicted remaining time to determine the work time for the request. To control the user-perceived delay, the server should compute its response within the work time.
There are several challenges in providing answers to the two API calls. Tracking elapsed time requires accurate and lightweight accounting across multiple, overlapping asynchronous activities that constitute the processing of a request on both the mobile device and the server. When the request reaches the server, we must also factor in the clock offset and drift between the mobile app and the server. Inference of this skew is hindered by the high variability in the delay of cellular network links. Estimating remaining time is difficult because it depends on many factors such as device type, network type, network provider, response size, and prior transfers between the client and server (which dictate the TCP window size at the start of the current transfer).

Timecard addresses these challenges and provides accurate estimates to the API calls. It consists of three main components:

- **Runtime User Transaction Tracking:** Timecard tracks user transactions at runtime. Timecard automatically instruments both the mobile app code, and the cloud service code to track the accumulated elapsed time, carrying this value across the stream of threads and asynchronous calls on both the mobile client and server.

- **Time Synchronization:** Timecard also includes a time synchronization method to accurately infer the clock drift and offset between the mobile app and the server. Our technique is similar to NTP but it is both radio-aware and network-aware. It sends probes only when the mobile network link is idle and stable thereby achieving high accuracy and incurring low overhead.

- **Remaining Time Prediction:** To predict the remaining time, Timecard separately predicts the network delivery time and the processing time at the app. Timecard trains and uses a decision-tree classifier to predict the network delivery time. It takes several relevant factors into account, including the intended response size, the round-trip time, the number of bytes already transferred on the connection prior to this response, and the network provider. Timecard estimates the processing time using a similar decision-tree model that takes the intended response size and device type into account.

We have implemented Timecard for the Windows Phone platform and .NET cloud services. The keys results are summarized below. These results suggest that Timecard is a practical and useful way to build cloud-based mobile apps with predictable response times.

- To study the effectiveness of Timecard, we modified two mobile services to adapt their response quality using the Timecard API. We instrumented the corresponding mobile apps and recruited 20 users to run them on their personal phones for over a month. This deployment yielded more than 300,000 transactions with various network conditions and device types. The results show that Timecard can tightly control the end-to-end response time around a desired user-perceived delay. For instance, the response time is within 50 ms of the desired bound (1200 ms) 90% of the time in one of the apps.

- To answer if Timecard is useful in practice, we analyzed the runtime of 4000 popular Windows Phone apps using the monkeys in VanarSena (without any fault inducer). We find that 80% of user interactions across 4000 popular Windows Phone apps involved network communication that could have benefited from Timecard.
• Similar to AppInsight, Timecard is light-weight. The runtime overhead is less than 0.1% in the mobile app and negligible at the server. The memory and network overhead is less than 1%.

1.5 Contributions and Roadmap

In addition to the design and deployment of systems to test, monitor, and adapt mobile apps, this dissertation makes the following technical contributions:

- **User Transactions**: It introduces the notion of user transactions in mobile apps. Tracking the execution of the app as a set of user transactions enables us to understand and reason about user-perceived performance and failures and allows us to efficiently test, monitor, and adapt the apps.

- **Managing user-perceived delays**: It presents novel techniques to improve and control user-perceived delays in mobile apps. AppInsight identifies performance bottlenecks and provides feedback to the developer about optimizations needed for improving user-perceived delays. Timecard reduces performance variability by enabling apps to tightly control user-perceived delays around a certain value.

- **Insight into real-world crashes and performance**: The systems presented in this dissertation provide detailed insights into real-world crashes and performance in mobile apps. VanarSena studies 25 million crash reports from the wild and uncovered bugs in thousands of apps deployed in the app store. The deployment in AppInsight and Timecard provides detailed insights into performance bottlenecks, components of delay, and performance variability in real apps.

The rest of this dissertation is organized as follows. Chapter 2 describes our instrumentation framework and techniques to do user transaction tracking. Chapter 3 describes the VanarSena system that does automated app testing to uncover failures before the app is deployed. Chapter 4 presents AppInsight, a system to monitor mobile apps after the app is deployed in the wild. Chapter 5 describes Timecard which enables mobile apps to provide consistent performance by adapting to variable conditions in the wild. Chapter 6 concludes the dissertation and outlines directions for future work.
Chapter 2

Tracking User Transactions

VanarSena, AppInsight, and Timecard presented in Chapters 3, 4, and 5 respectively, makes it easy for app developers to diagnose and improve their mobile apps. To minimize app developer effort, the systems are built on top of a binary framework that automatically instruments app binaries. In this chapter, we describe the binary instrumentation framework and the techniques to efficiently track app performance and failures. We first describe the typical asynchronous programming pattern used in mobile apps, and the challenge it presents for monitoring app execution. We then describe our instrumentation techniques to do user transaction tracking, a key component common to all three systems.

2.1 Mobile App Programming Pattern

Mobile apps are UI-centric in nature. The app user interface typically consists of a set of controls for user interaction and displaying content. Users interact with UI controls such as buttons to perform application tasks. Figure 2-1 shows screenshots from a sample mobile app. When the user clicks on the search button, the app fetches data from Twitter, processes the data, and updates the UI with the search result.

In modern UI programming frameworks [26, 71], the UI is managed by a single, dedicated thread. All UI updates, and all user interactions with the UI take place on this thread. To maintain UI responsiveness, applications avoid blocking the UI thread as much as possible, and perform most work asynchronously. Some mobile-programming frameworks like Silverlight [71], do not even provide synchronous APIs for time-consuming operations like network I/O and location queries. Even compute tasks are typically carried out by spawning worker threads. Thus, user requests are processed in a highly asynchronous manner.

This is illustrated in Figure 2-3, which shows the execution trace for a simple code snippet in Figure 2-2. In the figure, horizontal line segments indicate time spent in thread execution, while arrows between line segments indicate causal relationships between threads.

(0) The user starts the transaction by clicking a button; (1) the OS invokes the event handler (btnFetch_Click) in the context of the UI thread; (2) the handler makes an asynchronous HTTP request, providing reqCallback as the callback; (3) the handler quits, freeing the UI thread; (4) time is spent downloading the HTTP content; (5) when the HTTP request completes, the OS calls reqCallback in a worker thread; (6) the worker thread processes the fetched data; (7) when the processing finishes, the worker thread invokes the UI Dispatcher, to queue a UI update; (8) the OS calls the dispatched function (updateUI) asynchronously on the UI thread, which updates the UI.
Figure 2-1: User interface for a sample mobile app. When the user clicks on the search button, the app fetches data from Twitter, processes the data, and updates the UI with the search result.

```csharp
void btnFetch_Click(object sender, RoutedEventArgs e) {
    var req = WebRequest.Create(url);
    req.BeginGetResponse(reqCallback, null);
}

void reqCallback(IAsyncResult result) {
    /* Process result */
    UIDispatcher.BeginInvoke(updateUI);
}

void updateUI() {
    /* Update UI */
}
```

Figure 2-2: Asynchronous coding pattern in mobile apps.
The user-perceived delay for this interaction is the time between the user interaction and the UI update. To track the execution of this interaction and measure its performance, we need to instrument the app to track the entire execution graph in Figure 2-3 spanning multiple threads and asynchronous calls.

Real apps, of course, are much more complex. Figure 2-4 shows the CDF of number of edges in the execution graph for 167,000 tasks from 30 popular Windows Phone apps used by 30 users for over 6 months. There are execution graphs with more than 8000 edges. In real apps, (i) worker threads may in turn start their own worker threads, (ii) some user interactions may start a timer to perform periodic tasks through the lifetime of an app, (iii) transactions may be triggered by sensors such as accelerometers and, (iv) a user may interrupt a running transaction or start another one in parallel.

For example, Figure 2-5 illustrates a pattern common to location-based apps. The app displays information about nearby restaurants and attractions to the user. A typical user
transaction goes as follows. Upon user manipulation, the app asks the system to get a GPS fix, and supplies a callback to invoke when the fix is obtained. The system obtains the fix, and invokes the app-supplied callback in a worker thread at (2). The callback function reads the GPS coordinates and makes two parallel web requests to fetch some location-specific data. Then, the thread waits (4), for two completion signals. The wait is indicated via a dotted line. As the two web requests complete, the OS invokes their callbacks at (5) and (7). The first callback signals completion to the blocked thread at (6), while the second one does it at (8). As a result of the second signal, the blocked thread wakes up at (9), and updates the UI via the dispatcher.

Given such complex behavior, it can be difficult for the developers to ascertain where the bottlenecks in the code are and what optimizations might improve user-perceived responsiveness. In Figure 2-5, the bottleneck path involves the second web request, which took longer to complete. Worse, these bottlenecks may be different for different users, depending on their device, location, network conditions, and usage patterns.

Failure analysis is also complicated by the asynchronous nature of the app. Consider the example in Figure 2-6. Suppose the app crashes in the method `parseURL()` (8), which is called in a worker thread that started at `parseXML()` (7). Since the UI thread function that started the web request has exited, the OS has no information about the user context for this crash. Thus, in the exception log offered by today's popular mobile platforms, the developer will only see the stack trace of the crashed thread, from `parseURL()` to `parseXML()`. The developer however, might want more information, such as the user manipulation that triggered the crash, to speed up debugging. This underscores the need for a system that can track user transactions across thread boundaries.

A user transaction begins with a user interaction of the UI, and ends with completion of all synchronous and asynchronous tasks (threads) in the app that were triggered by the interaction. For example, in Figure 2-3, the user transaction starts when the user interaction occurs and ends when the `updateUI` method completes. To monitor app performance and failures, our goal is to instrument the app to track its execution as a set of user transactions.
Before we present our instrumentation techniques (§2.3), we describe our binary instrumentation framework.

### 2.2 Binary Instrumentation of Mobile Apps

Mobile apps are often written using higher-level programming frameworks and compiled to an intermediate language (bytecode). For instance, Windows Phone apps are written using the Silverlight framework [71] in C#, compiled to MSIL [57] bytecode. MSIL is a stack-based language. MSIL preserves the structure of the program, including types, methods and inheritance information. Figure 2-7 shows the MSIL code for the `btnFetch_Click` method in Figure 2-2. Silverlight is used by a vast majority of the apps in the Windows Phone app store.

We built a binary instrumentation framework for Windows Phone apps to instrument at the bytecode level. The instrumentation framework can unpack the app binary, inject new bytecode or rewrite existing bytecode, and repack the binary without any access to source code and without any developer support. Instrumentation requires no special support from the Silverlight framework either. Figure 2-8 shows instrumentation of MSIL bytecode. The start and end of the `btnFetch_Click` method is instrumented to call external tracker methods.

In addition to rewriting app methods, the instrumentation framework can be used to add new libraries to the app binary, add event handlers to monitor app events, and track unhandled exceptions. We use this instrumentation framework to automatically track user transactions, which we turn to next.

### 2.3 Instrumentation to Track User Transactions

We now describe our instrumentation techniques in detail. Our goal is to track app execution as a set of user transactions. In VanarSena (§3), we track the start and completion of user transactions.
```csharp
.method private hidebysig instance void btnFetch_Click(object sender,
{
    .maxstack 4
    .locals init (     
    L_0000: ldarg.0
    L_0001: ldfld string WebRequestILTest.MainPage::url
    L_000b: stloc.0
    L_000c: ldloc.0
    L_000d: ldarg.0
    L_000e: ldfn instance void
           WebRequestILTest.MainPage::reqCallback(
               class [mscorlib]System.IAsyncResult)
    L_0014: newobj instance void
            [mscorlib]System.AsyncCallback::.ctor(object, native int)
    L_0019: ldnull
    L_001a: callvirt instance class [mscorlib]System.IAsyncResult
                class [mscorlib]System.AsyncCallback, object)
    L_001f: pop
    L_0020: ret
}

Figure 2-7: MSIL code for the btnFetch_Click method in Figure 2-2.
Figure 2-8: Instrumentation of MSIL bytecode. The start and end of the btnFetch_Click method is instrumented to call external tracker methods. A static identifier (5) is passed to the tracker methods.
transactions to efficiently test the app. In AppInsight (§4), we capture user transactions in the wild to provide feedback about user-perceived delays, performance bottlenecks, and failures. In Timecard (§5), we track user transactions at runtime to expose the delays of different components in a transaction to the application so that it can adapt.

In this section, we describe user transaction tracking mainly in the context of AppInsight (§4), where our goal is to capture, with minimal overhead, the information necessary to build execution traces of user transactions. Capturing user transactions enables us to measure user-perceived performance, find bottlenecks, and reason about failures. The transaction tracking techniques in VanarSena (§3) and Timecard (§5) are direct extensions of the ideas presented here. We explain the differences in the respective chapters.

The app-execution trace can be captured in varying degrees of detail. In deciding what to capture, we must strike the right balance between the overhead and our ability to provide useful feedback about performance and failures. Figures 2-5 and 2-6 indicate that, we need to track six categories of data: (i) when the user interacts with the UI; (ii) when the app code executes on various threads (i.e., start and end of horizontal line segments); (iii) causality between asynchronous calls and callbacks; (iv) thread synchronization points (e.g., through Wait calls) and their causal relationship; (v) when the UI is updated; (vi) any unhandled exceptions.

To capture user transactions, we instrument the app in two steps. First, we link two libraries to the app – a Detour library and a Tracker library (see Figure 2-9). The Detour library is dynamically generated during instrumentation. It exports a series of detouring functions [47], which help attribute callback executions to the asynchronous calls that triggered them. In the context of AppInsight, the Tracker library exports several logging methods to insert log records. In the context of VanarSena and Timecard, the Tracker library keeps data structures to monitor the transaction at runtime.

Second, we instrument the app binary with calls to methods in the Tracker and Detour libraries from appropriate places to collect the data we need. Below, we describe this process in detail. We use the code fragment shown in Figure 2-2, and the corresponding transaction diagram in Figure 2-3 as a running example.
2.3.1 Tracking User Interaction

When the user interacts with the UI (touch, flick, etc.) the Silverlight framework delivers several UI input events on the UI thread of the app running in the foreground. The first event in this series a ManipulationStarted event, and the last is the ManipulationEnded event. Further, any app-specified handler to handle the UI event is also called on the UI thread in between these two events. For example, in Figure 2-2, btnFetch_Click handles the click event for a button. When the user touches the button on the screen, the handler is called in between the two Manipulation events.

The Tracker library exports handlers for ManipulationStarted and ManipulationEnded events, which we add to the app’s code. This allows us to match the UI interaction to the right app handler for that UI input.

2.3.2 Tracking thread execution

The horizontal line segments in Figure 2-3 indicate when the app code starts and ends executing on each thread. This can be determined from a full execution trace that logs the start and end of every method. However, the overhead of capturing a full execution trace from a mobile phone is prohibitive. We reduce the overhead substantially by observing that at the beginning of each horizontal line segment in Figure 2-3, the top frame in the thread’s stack corresponds to an app method (as opposed to a method that is internal to the framework) and that this method is the only app method on the stack. These methods are upcalls from the framework into the app code. To capture user transactions, it is enough to track the start and end of only upcalls.

The upcalls are generated when the system invokes an app-specified handler (also called callback) methods for various reasons, for example, to handle user input, timer expiration, sensor triggers, or completion of I/O operations. Even spawning of worker threads involves upcalls: the app creates a thread, and specifies a method as a start method. This method is invoked as a callback of Thread.Start at some later time.

We identify all potential upcall methods using two simple heuristics. (i) Event handlers attached to UI controls are upcalls: In Silverlight, UI event handlers have a well-defined signature. We consider methods with that signature to be an upcall. (ii) Function pointers are potential upcalls: When a method is specified as a callback to a system call, a reference to it, a function pointer, called delegate in .NET parlance, is passed to the system call. For example, in Figure 2-2, a reference to reqCallback is passed to the BeginGetResponse system call. The MSIL code for creating a delegate has a fixed format [57], in which two special opcodes are used to push a function pointer onto the stack. Any method that is referenced by these opcodes may be called as an upcall.

We track the start and end of all potential upcalls, as shown in Figure 2-10. The instrumentation added for tracking potential upcalls is prepended by ‘+’. All three methods in the example are potential upcalls and thus instrumented. The method btn_FetchClick is a UI handler, and the other two methods are passed as callbacks. While this technique is guaranteed to capture all upcalls, it may instrument more methods than necessary, imposing unnecessary overhead. But we find the overhead to be negligible (§4.6.6).

We instrument the start and end of upcalls to call into the Tracker methods with three parameters:

- **Thread id**: The current thread id is used to link the asynchronous calls made during the execution of the thread to the transaction. It is also used to the link the
asynchronous callback invocation to the upcall that is invoked.

- **Upcall match id:** A match id is dynamically generated to match the start and end of the upcall.

- **Method id:** During instrumentation, a static method id is generated for each method (5 for btnFetch_click in Figure 2-10). A mapping between the method id and the method name is maintained in a separate manifest. Tracking the method id helps us to provide the method name in the feedback to developers.

### 2.3.3 Matching async calls to their callbacks

We described how we instrument all methods that may be used as upcalls. We now describe how we match asynchronous calls to the resulting upcalls (i.e., their callbacks). For example, in Figure 2-3, we need to match labels 2 and 5. To do so, we need to solve three problems.

First, we need to identify all call sites where an asynchronous system call is made, e.g., in Figure 2-2, the BeginGetResponse call is an asynchronous system call. Second, we need to track when the callback starts executing as an upcall. Third, we need to connect the beginning of callback execution to the right asynchronous call.

We solve the first problem by assuming platform knowledge. We know the list of asynchronous calls in the Silverlight framework and we can identify them during instrumentation. We have already described how to solve the second problem by tracking the start of upcall execution. The third problem of connecting the callback to the right asynchronous call is a challenging one. This is because a single callback function (e.g., a completion handler for a web request) may be specified as a callback for several asynchronous system calls. One possibility is to rewrite the app code to clone the callback function several times, and assign them unique ids. However, this is not sufficient, since the asynchronous call may be called in a loop (e.g., for each URL in a list, start download) and specify the same function as a callback. To handle such scenarios, we rewrite the callback methods to detour them through the Detour library, as described below.

Figure 2-10 shows instrumented code for the example in Figure 2-2. Instrumentation used for detour is tagged with '.*'. Figure 2-11 shows relevant code inside the Detour library. We add instrumentation as follows.

(i) We identify the system call BeginGetResponse as an asynchronous call.

(ii) We generate a new method called cb1 that matches the signature of the supplied callback function, i.e., reqCallback, and add it to the Detour class in the Detour library. This method is responsible for invoking the original callback (see Figure 2-11).

(iii) We instrument the call site to generate a new instance of the Detour object. This object stores the original callback, and is assigned a unique id (called asyncMatchId) at runtime. This asyncMatchId helps match the asynchronous call to the callback.

(iv) We then rewrite the app code to replace the original callback argument with the newly generated detour method, Detour.cb1.

Notice from Figure 2-11 that we track the beginning of an asynchronous call using the TrackAsyncCall function of the Tracker library. The parameters to the call include:

- **Thread id:** The thread id is used to link this asynchronous call to its containing upcall.
void btnFetch_Click(object sender, RoutedEventArgs e) {
    + int threadId = Thread.CurrentThread.ManagedThreadId;
    + int upcallMatchId = Tracker.GenerateUpcallId();
    + Tracker.TrackUpcallStart(threadId, upcallMatchId, 5);
    var req = WebRequest.Create(url);
    * int asyncMatchId = Tracker.GenerateAsyncId();
    * Detour dt = new Detour(reqCallback, asyncMatchId);
    * Tracker.TrackAsyncCall(threadId, asyncMatchId, 16);
    req.BeginGetResponse(dt.Cbl, null);
    + Tracker.TrackUpcallEnd(threadId, upcallMatchId, 5);
}

void reqCallback(IAsyncResult result) {
    + int threadId = Thread.CurrentThread.ManagedThreadId;
    + int upcallMatchId = Tracker.GenerateUpcallId();
    + Tracker.TrackUpcallStart(threadId, upcallMatchId, 19);
    /* Process */
    * int asyncMatchId = Tracker.GenerateAsyncId();
    * Detour dt = new Detour(updateUI, asyncMatchId);
    * Tracker.TrackAsyncCall(asyncMatchId, 7);
    UIDispatcher.BeginInvoke(dt.Cb2);
    + Tracker.TrackUpcallEnd(threadId, upcallMatchId, 19);
}

void updateUI() {
    + int threadId = Thread.CurrentThread.ManagedThreadId;
    + int upcallMatchId = Tracker.GenerateUpcallId();
    + Tracker.TrackUpcallStart(threadId, upcallMatchId, 21);
    /* Update UI */
    + Tracker.TrackUpcallEnd(threadId, upcallMatchId, 21);
}

Figure 2-10: Instrumented version of the code in Figure 2-2. The actual instrumentation is done on MSIL byte code. We show decompiled C# code for convenience.
public class Detour {
    Delegate originalCb;
    int asyncMatchId;

    public Detour(Delegate d, int matchId) {
        this.originalCb = d;
        this.asyncMatchId = matchId;
    }

    public void Cb1(IAsyncResult result) {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Tracker.TrackAsyncCallback(threadId, this.asyncMatchId);
        Invoke(this.originalCb);
    }

    public void Cb2() {
        ...
    }
}

Figure 2-11: Detour library

- **Async match id**: The dynamically generated match id stored in the Detour object is used to link the asynchronous call to its callback.

- **Call id**: Similar to the method id, the call id is used to map back to code for providing developer feedback.

The beginning of the callback is tracked by the TrackAsyncCallback, which is called from cb1, just before the original callback is invoked. The parameters to the call include:

- **Thread id**: The thread id is used to link the callback invocation to the invoked upcall.

- **Async match id**: The match id stored in the detour object is used to link the asynchronous call to its callback.

TrackAsyncCall, TrackAsyncCallback, and the UpcallStart of the original callback method are linked by the asyncMatchId and their thread ids, allowing us to attribute the callback to the right asynchronous call. We show an example in Figure 2-14 and §2.3.7.

Figure 2-10 also shows another example of detouring. The UpdateUI method is a callback for the BeginInvoke method of the UIDispatcher, and hence is detoured.

Delayed Callbacks
The above technique to match asynchronous call and callback needs refinement in practice. Consider Figure 2-12. The callback delegate foo was specified when the constructor was called, but it is invoked only when Thread.Start is called, which may be much later. Our technique would incorrectly match the callback to the call site of the constructor, instead
public class AppWorker {

    Thread t;

    public AppWorker() {
        t = new Thread(foo);
    }

    public void Process() {
        t.Start();
    }
}

Figure 2-12: Delayed callback

of Thread.Start. We use domain knowledge about Silverlight system libraries to solve this problem. We know that the callback function is always invoked from Thread.Start. To correctly match the async call, we track the invoked objects at these call sites. Our instrumentation is shown in Figure 2-13. After detouring the callback at the Thread constructor, we call the MapObjectDetour method to store a mapping between the created thread object and the detour object. Before the Thread.Start call, we retrieve the detour object for the given thread object. Note that, the thread object is the same after the constructor and before the Thread.Start call. We now call TrackAsyncCall with the asyncMatchId in the retrieved detour object. This allow us to match the callback to the Thread.Start call as before. We handle event subscriptions in a similar manner.

2.3.4 Tracking Thread Synchronization

Silverlight provides a set of methods for thread synchronization. A thread can wait on a synchronization object (semaphore) using the Monitor.Wait(syncObject) call. The thread is blocked until the synchronization object is signaled using the Monitor.Pulse(syncObject) call from another thread. The causality edge between these threads starts at the Monitor.Pulse call and ends at the return of the Monitor.Wait call (i.e. when the thread is unblocked). This is illustrated in Figure 2-15. To track the thread synchronization edges, we instrument calls to Monitor.Wait and Monitor.Pulse, and track the identities of the synchronization objects they use.

In Silverlight, a thread can wait on multiple objects to be signaled using the Monitor.WaitAll(syncObjects) call. Similarly, Monitor.WaitAny(syncObjects) can be used to wait until at least one object is signaled. Figure 2-16 shows an example where the main thread waits for two processing threads to complete. Figure 2-17 shows the instrumented code. We instrument start and end of the wait call and start of the pulse call. As shown in Figure 2-18, we are able to infer the causality between the threads by tracking their thread ids and identities of the synchronization objects used.

Thread join calls are handled similarly by tracking the identity of the thread objects, and by tracking the completion of the thread callbacks.
public class AppWorker {

    Thread t;

    public AppWorker() {
        * int asyncMatchId = Tracker.GenerateAsyncId();
        * Detour dt = new Detour(foo, asyncMatchId);
        t = new Thread(foo);
        * Tracker.MapObjectDetour(t, dt);
    }

    public void Process() {
        * Detour dt = Tracker.GetDetour(t);
        * Tracker.TrackAsyncCall(threadId, dt.asyncMatchId, 32);
        t.Start();
    }
}

Figure 2-13: Instrumentation for delayed callbacks.

Figure 2-14: Capturing User Transactions.
Figure 2-15: Thread synchronization and causal edges.

Figure 2-16: Thread synchronization example where the main thread waits for two processing threads to complete.
public class ParallelProcess {

    Object syncObj1 = new Object();
    Object syncObj2 = new Object();

    public void Run() {
        /* Asynchronously call Process1() */
        /* Asynchronously call Process2() */
        Object[] syncObjects = new Object[] { syncObj1, syncObj2 };
        * Tracker.TrackWaitAllCallStart(threadId, syncObjects, 27);
        Monitor.WaitAll(syncObjects);
        * Tracker.TrackWaitAllCallEnd(threadId, 27);
        /* Both processing is complete */
    }

    public void Process1() {
        /* Processing 1 */
        * Tracker.TrackPulseCall(threadId, syncObj1, 41);
        Monitor.Pulse(syncObj1);
    }

    public void Process2() {
        /* Processing 2 */
        * Tracker.TrackPulseCall(threadId, syncObj2, 44);
        Monitor.Pulse(syncObj2);
    }
}

Figure 2-17: Instrumented code for the thread synchronization example in Figure 2-16.
2.3.5 Tracking UI updates

To understand user-perceived performance, it is important to monitor when UI updates happen. The Silverlight framework generates a special LayoutUpdated event whenever the app finishes updating the UI. Specifically, if an upcall runs on the UI thread (either event handlers, or app methods called via the UIDispatcher), and updates one or more elements of the UI as part of its execution, then a LayoutUpdated event is raised when the upcall ends. The Tracker library exports a handler for this event, which we add to the app code. Using the LayoutUpdated events, we can correctly identify the threads that update the UI. This enables us to accurately measure the user-perceived performance.

2.3.6 Tracking unhandled exceptions

When an unhandled exception occurs in the app code, the system terminates the app. Before terminating, the system delivers a special unhandled exception event to the app. The data associated with this event contains the exception type and the stack trace of the thread in which the exception occurred. To log this data, the Tracker library exports a handler for this event, which we add to the app code. Using the transaction log and the exception trace, we can build detailed failure traces as shown in Figure 2-6, which we call the exception path (§4).

2.3.7 Example trace

Table 2.1 shows the trace of tracker calls generated by the instrumented code in Figure 2-10. Records 1 and 5 show UI Manipulation events. They encompass an upcall (records 2-4) to the method btnFetch_Click. As described in §2.3.1, we attribute this upcall to UI interaction.
This method makes the asynchronous system call `BeginGetResponse` (record 3), the callback of which is detoured, and assigned an async match id of 1001. Record 6 marks the beginning of the execution of the detoured callback. It calls the actual callback method, `reqCallback`, which has a upcall match id of 502. This method executes between records 7 and 9. We can link records 6 and 7 because they have the same thread id, and will always follow each other (§2.3.3). When `reqCallback` executes, it makes another asynchronous call. This is the call to the UI dispatcher. We detour the callback, and assign it a async match id of 1002. The actual callback method, of course, is `UpdateUI`, which has the upcall match id of 503.

The completion of this method is indicated by record 12. We note that this method ran on the UI thread. Record 12 indicates that a `LayoutUpdated` event was triggered immediately after the execution of this method, which means that this method must have updated the UI. Thus, we can reconstruct the complete user transaction.

Systems described in the next three chapters use the binary instrumentation framework and the notion of user transactions to improve the reliability and performance of apps.

### 2.4 Supporting Other Platforms

The current implementation of user transaction tracking focuses on Windows Phone apps and the Silverlight platform. However, these techniques can be ported to other platforms as well. The core ideas behind instrumentation and user transaction tracking can be applied to any platform that has following basic characteristics:

- The app binary has a (byte) code representation that preserves the structure of the source code, including types, classes, methods, and inheritance information.
- The platform has statically identifiable asynchronous calls.

<table>
<thead>
<tr>
<th>Tracker Events</th>
<th>ThreadId</th>
<th>upcallMatchId</th>
<th>asyncMatchId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 UIManipulationStarted</td>
<td>0</td>
<td>501</td>
<td>1001</td>
</tr>
<tr>
<td>2 UpcallStart(5)</td>
<td>0</td>
<td>501</td>
<td></td>
</tr>
<tr>
<td>3 AsyncCall(16)</td>
<td>0</td>
<td>501</td>
<td></td>
</tr>
<tr>
<td>4 UpcallEnd(5)</td>
<td>0</td>
<td>501</td>
<td></td>
</tr>
<tr>
<td>5 UIManipulationEnded</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 AsyncCallback</td>
<td>1</td>
<td>1001</td>
<td>1002</td>
</tr>
<tr>
<td>7 UpcallStart(19)</td>
<td>1</td>
<td>502</td>
<td>1002</td>
</tr>
<tr>
<td>8 AsyncCall(7)</td>
<td>1</td>
<td>502</td>
<td></td>
</tr>
<tr>
<td>9 UpcallEnd(19)</td>
<td>1</td>
<td>503</td>
<td></td>
</tr>
<tr>
<td>10 AsynCallback</td>
<td>0</td>
<td></td>
<td>1002</td>
</tr>
<tr>
<td>11 UpcallStart(21)</td>
<td>0</td>
<td>503</td>
<td></td>
</tr>
<tr>
<td>12 UpcallEnd(21)</td>
<td>0</td>
<td>503</td>
<td></td>
</tr>
<tr>
<td>13 LayoutUpdated</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Trace of code in Fig. 2-10. The UI thread id is 0. Method id and call id are shown within parenthesis.
- It supports subscription to events that clearly identifies user interactions and updates to the UI.

- It has well-defined thread synchronization primitives.

These requirements are not onerous. SIF [44] is a recent effort that has ported the techniques described in this chapter to Android [13]. Majority of apps in the Android app store are written in Java which compiles down to Java byte code. The Android platform and the Java byte code satisfies all the properties discussed above and hence it is relatively easy to port the tracking techniques. Apps for iOS [48] and certain games for Android and Windows Phone are written in C or C++ and they compile down to native code. While byte code preserves the source code structure, native code does not, and hence binary instrumentation is harder. For such apps, instead of doing binary instrumentation as a post-build step, we could do source code instrumentation as a pre-build step. The Instrumenter can analyze and instrument the source code before it is compiled to native code.
Chapter 3

Automated App Testing

Mobile app testing poses different challenges than traditional software. Mobile apps run across a wide range of environmental conditions, locations, and hardware platforms. They are often subject to an array of interactions, input data, and sensor behavior. Coping with these issues is particularly acute for individual developers or small teams.

Our goal is to develop an easy to use, and scalable system that thoroughly tests mobile apps for common failures before the app is deployed to users. The developer should be able to submit an app binary to the system, and then within a short amount of time obtain a report. This report should provide a correct stack trace and a trace of interactions or inputs for each failure. We anticipate the system being used by developers interactively while debugging, as well as a part of regular nightly and weekly regression tests, so speed is important. An ideal way to deploy the system is as a service in the cloud, so the ability to balance resource consumption and discovering faults is also important.

There are various approaches for mobile app testing. Static analysis of app binaries [58, 30], although scalable, can fail to uncover app faults due to the runtime issues such as poor network condition and corrupted or unexpected responses from cloud services. Symbolic execution [20] and its hybrid variant, concolic execution, require constructing symbolic model for program execution environment. Although such a model is shown feasible for simple android libraries [62] and UI events [12, 34] of simple apps, applicability of the techniques have been limited for two key reasons. First, it is not easy to model real-world execution environment for mobile apps, consisting of sensors, networks, and cloud services. Second, they do not scale well to real-world apps due to notorious path explosion problem. Recent efforts, therefore, focus on dynamic analysis, where runtime behavior of an app is examined by executing it [74, 29, 14, 55, 59, 56]. We take a similar approach.

In this chapter, we describe VanarSena, a system that does automated app testing by systematically emulating user interactions and fault conditions from the wild to uncover app failures. The starting point in the design of VanarSena is to identify what types of faults have the highest “bang for the buck” in terms of causing real-world failures. To this end, we studied 25 million crash reports from more than 100,000 Windows Phone apps reported in 2012. Three key findings inform our design: first, over 90% of the crashes were attributable to only 10% of all the root causes we observed. Second, although the “90-10” rule holds, the root causes affect a wide variety of execution paths in an app. Third, a significant fraction of these crashes can be mapped to externally induced events, such as unhandled HTTP error codes (see §3.1).

The first finding indicates that focusing on a small number of root causes will improve
reliability significantly. The second suggests that the fault finder needs to cover as many execution paths as possible. The third indicates that software emulation of user inputs, network behavior, and sensor data is likely to be effective, even without deploying on phone hardware.

In VanarSena, the developer uploads the app binary to the service, along with any supporting information such as a login and password. VanarSena instruments the app, and launches several *monkeys* to run the instrumented version on phone emulators\(^1\). As the app is running, VanarSena emulates a variety of user, network and sensor behaviors to uncover and report observed failures.

A noteworthy principle in VanarSena is its “greybox” approach, which instruments the app binary before emulating its execution. Greybox testing combines the benefits of “whitebox” testing, which requires detailed knowledge of an app’s semantics to model interactions and inputs, but isn’t generalizable, and “blackbox” testing, which is general but not as efficient in covering execution paths.

The use of binary instrumentation enables a form of execution-path exploration we call *hit testing* (§3.5.1), which identifies how each user interaction maps to an event handler and in turn a user transaction. Hit testing allows VanarSena to cover many more execution paths in a given amount of time. Binary instrumentation also allows VanarSena to track the completion of user transactions and determine when to emulate the next user interaction in the app. This task is tricky because emulating a typical user requires knowing when the previous interaction has been processed and rendered. We call this generation of *ProcessingCompleted* event (§3.5.2), which directly leverages the techniques in Chapter 2. Moreover, app instrumentation makes VanarSena extensible, by inserting our own event handlers that trigger under certain situations, such as network calls and certain user actions. VanarSena can then trap these event handlers to induce specific faults such as emulating slow or faulty networks. We have written several such fault inducers, and more can be easily written.

We have implemented VanarSena for Windows Phone apps, running it as an experimental service. We evaluated VanarSena empirically by testing 3,000 apps from the Windows Phone store for commonly-occurring faults. VanarSena discovered failures in 1,108 of these apps, which have presumably undergone some testing and real-world use\(^2\). Overall, VanarSena detected 2,969 crashes, including 1,227 that were not previously reported. The testing took 4500 machine hours on 12 desktop-class machines, at average of 1.5 hours per app. At current Azure prices, the cost of testing is roughly 25 cents per app. These favorable cost and time estimates result from VanarSena’s use of *hit testing* and *ProcessingCompleted* event.

### 3.1 App Crashes in-the-Wild

To understand why apps crash in the wild, we analyze a large data set of crash reports. We describe our data set, our method for determining the causes of crashes, and the results of the analysis.

---

\(^1\) *VanarSena* in Hindi means an “army of monkeys”.

\(^2\) Thus, VanarSena would be even more effective during earlier stages of development.
Figure 3-1: CDF of crash reports per app.

0: TransitTracker.BusPredictionManager.ReadCompleted
...

Figure 3-2: Stack trace fragment for Chicago Transit Tracker crash. The exception was WebException.

3.1.1 Data Set

Our data set was collected by Windows Phone Error Reporting (WPER) system, a repository of error reports from all deployed Windows Phone apps. When an app crashes due to an unhandled exception, the phone sends a crash report to WPER with a small sampling probability\(^3\). The crash report includes the app ID, the exception type, the stack trace, and device state information such as the amount of free memory, radio signal strength, etc.

We study over 25 million crash reports from more than 100,000 apps collected in 2012. Figure 3-1 shows the number of crash reports per app. Observe that the data set is not skewed by crashes from handful of bad apps. A similar analysis shows that the data is not skewed by a small number of device types, ISPs, or countries of origin.

3.1.2 Root Causes of Observed Crashes

To determine the root cause of a crash, we start with the stack trace and the exception type. An exception type gives a general idea about what went wrong, while the stack trace indicates where things went wrong. An example stack fragment is shown in Figure 3-2. Here, a WebException was thrown, indicating that something went wrong with a web transfer, causing the OnOpenReadCompleted function of the WebClient class to throw an exception. The exception surfaced in the ReadCompleted event handler of the app, which did not handle it, causing the app to crash.

We partition crash reports that we believe originate due to the same root cause into a collection called a crash bucket: each crash bucket has a specific exception type and system

\(^3\)The developer has no control over the probability.
function name where the exception was thrown. For example, the crash shown in Figure 3-2 will be placed in the bucket labeled WebException, System.Net.WebClient.OnOpenReadCompleted.

Given a bucket, we use two techniques to determine the likely root cause of its crashes. First, we use data mining techniques [7] to discover possible patterns of unusual device states (such as low memory or poor signal strength) that hold for all crashes in the bucket. For example, we found that all buckets with label (OutOfMemoryException, *) have the pattern AvailableMemory = 0.

Second, given a bucket, we manually search various Windows Phone developer forums such as social.msdn.microsoft.com and stackoverflow.com for issues related to the exception and the stack traces in the bucket. We limit such analysis to only the 100 largest buckets, as it is not practical to investigate all buckets and developer forums do not contain enough information about less frequent crashes. We learned enough to determine the root causes of 40 of the top 100 buckets. We also manually verified the root causes we determined. The whole process took us around one week.

3.1.3 Findings

A small number of large buckets cover most of the crashes. Figure 3-3 shows the cumulative distribution of various bucket sizes. The top 10% buckets cover more than 90% crashes (note the log-scale on the x-axis). This suggests that we can analyze a small number of top buckets and still cover a large fraction of crashes. Table 3.1 shows several large buckets of crashes.

A significant fraction of crashes can be mapped to well-defined externally-inducible root causes. We use the following taxonomy to classify various root causes. A root cause is deterministically inducible if it can be reproduced by deterministically modifying the external factors on which the app depends. For example, crashes of a networked app caused by improperly handling an HTTP Error 404 (Not Found) can be induced by an HTTP proxy that returns Error 404 on a Get request. Some crashes such as those due to memory faults or unstable OS states are not deterministically inducible. We further classify inducible causes into two categories: device and input. Device-related causes can be induced by systematically manipulating device states such as available memory, available storage,
Figure 3-4: Distinct stack traces in various buckets for one particular app

Figure 3-5: Distinct stack traces in various buckets for all apps
network signal, etc. Input-related causes can be induced by manipulating various external inputs to apps such as user inputs, data from network, sensor inputs, etc.

Table 3.1 shows several top crash buckets, along with their externally-inducible root causes and their categories. For example, the root causes behind the bucket with label \((\text{WebException, WebClient.OnDownloadStringCompleted})\) are various HTTP Get errors such as 401 (Unauthorized), 404 (Not Found), and 405 (Method Not Allowed), and can be induced with a web proxy intercepting all network communication to and from the app.

We were able to determine externally-inducible root causes of 40 of the top 100 buckets; for the remaining buckets, we either could not determine their root causes from information in developer forums or identify any obvious way to induce the root causes. Together, these buckets represent around 48% of crashes in the top 100 buckets (and 35% of all crashes); the number of unique root causes for these buckets is 8.

These results imply that a significant number of crashes can be induced with a relatively small number of root causes.

Although a small number, the dominant root causes affect many different execution paths in an app. For example, the same root cause of HTTP Error 404 can affect an app at many distinct execution points where the app downloads data from a server. To illustrate how often it happens, we consider all crashes from one particular app in Figure 3-4 and count the number of distinct stack traces in various crash buckets of the app. The higher the number of distinct stack traces in a bucket, the more the distinct execution points where the app crashed due to the same root causes responsible for the bucket. As shown in Figure 3-4, for 25 buckets, the number of distinct stack traces is more than 5. The trend holds in general, as shown in Figure 3-5, which plots the distribution of distinct stack traces in all (app, bucket) partitions. We find that it is common for the same root cause to affect many tens of execution paths of an app.
Table 3.1: Examples of crash buckets and corresponding root causes, categories, and ways to induce the crashes

<table>
<thead>
<tr>
<th>Rank (Fraction)</th>
<th>Exception</th>
<th>Bucket</th>
<th>Crash Function</th>
<th>Root Cause</th>
<th>Category</th>
<th>HowToInduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (7.51%)</td>
<td>OutOfMemory</td>
<td>*</td>
<td></td>
<td>WritablePages = 0</td>
<td>Device/Memory</td>
<td>Memory pressure</td>
</tr>
<tr>
<td>2 (6.09%)</td>
<td>InvalidOperation</td>
<td>ShellPageManager.</td>
<td>CheckHRESULT</td>
<td>User clicks buttons or links in quick succession, and thus tries to navigate to a new page when navigation is already in progress</td>
<td>Input/User</td>
<td>Impatient user</td>
</tr>
<tr>
<td>3 (5.24%)</td>
<td>InvalidOperation</td>
<td>NavigationService.</td>
<td>Navigate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 (2.66%)</td>
<td>InvalidOperation</td>
<td>NavigationService.</td>
<td>GoForwardBackCore</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 (1.16%)</td>
<td>WebException</td>
<td>Browser.AsyncHelper.</td>
<td>BeginOnUI</td>
<td>Unable to connect to remote server</td>
<td>Input/Network</td>
<td>Proxy</td>
</tr>
<tr>
<td>15 (0.83%)</td>
<td>WebException</td>
<td>WebClient.OnDownloadStringCompleted</td>
<td>HTTP errors 401, 404, 405</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 (2.30%)</td>
<td>XmlException</td>
<td>*</td>
<td></td>
<td>XML Parsing Error</td>
<td>Input/Data</td>
<td>Proxy</td>
</tr>
<tr>
<td>11 (1.14%)</td>
<td>NotSupportedException</td>
<td>XmlTextReaderImpl.</td>
<td>ParseDoctypeDecl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37 (0.42%)</td>
<td>FormatException</td>
<td>Double.Parse</td>
<td></td>
<td>Input Parsing Error</td>
<td>Input/User, Input/Data</td>
<td>Proxy</td>
</tr>
<tr>
<td>50 (0.35%)</td>
<td>FormatException</td>
<td>Int32.Parse</td>
<td></td>
<td>Input Parsing Error</td>
<td>Input/Data</td>
<td>Proxy</td>
</tr>
</tbody>
</table>
3.2 Goals and Non-Goals

Our goal is to build a scalable, easy to use system that tests mobile apps for common, externally-inducible faults as thoroughly as possible. We want to return the results of testing to the developer as quickly as possible, and for the system to be deployable as a cloud service in a scalable way.

VanarSena does not detect all app failures. For example, VanarSena cannot detect crashes that result from hardware idiosyncrasies, or failures caused by specific inputs, or even failures caused by the confluence of multiple simultaneous faults that we do test for. VanarSena also cannot find crashes that result from erroneous state maintenance; for example, an app may crash only after it has been run hundreds of times because some log file has grown too large. VanarSena cannot adequately test apps and games that require complex free-form gestures or specific order of inputs.

Before we describe the architecture of VanarSena, we need to discuss how we measure thoroughness, or coverage. Coverage of testing tools is traditionally measured by counting the fraction of basic blocks [15] of code they cover. However, this metric is not appropriate for our purpose. Mobile apps often include third party libraries of UI controls (e.g., fancy UI buttons). Most of the code in these libraries is inaccessible at run time, because the app typically uses only one or two of these controls. Thus, coverage, as measured by basic blocks covered would look unnecessarily poor.

Instead, we focus on the user-centric nature of mobile apps. A mobile app is typically built as a collection of pages. An example app called AroundMe is shown in Figure 3-6. The user navigates between pages by interacting with controls on the page. For example, each category listing on page 1 is a control. By clicking on any of the business categories on page 1, the user would navigate to page 2. Page 1 also has a swipe control. By swiping on the page, the user ends up on the search page (page 4). From a given page, the user can navigate to the parent page by pressing the back button. The navigation graph of the app is shown in Figure 3-7. The nodes of the graph represent pages, while the edges represent unique user transactions that cause the user to move between pages. Thus, we measure coverage in terms of unique pages visited [14], and unique user transactions mimicked by the tool. In §3.7.2, we will show that we cover typical apps as thoroughly as a human user.
Figure 3-6: Example app pages. UI elements pointed by red arrows can be interacted with. Arrows and numbers will be explained in Section 3.5.1.
3.3 Architecture

Figure 3-8 shows the architecture of VanarSena. VanarSena instruments the submitted app binary. The *Monkey manager* then spawns a number of *monkeys* to test the app. A *monkey* is a UI automation tool built around the Windows Phone Emulator. The monkey can automatically launch the app in the emulator and interact with the UI like a user. When the app is *monkeyed*, we systematically feed different inputs and emulate various faults. If the app crashes, the monkey generates a detailed crash report for the developer. Figure 3-9 shows the key components of the monkey.

**Emulator:** We use an off-the-shelf Windows Phone emulator in our implementation. We intentionally do not modify the emulator in any way. The key benefit of using an emulator instead of device hardware is scalability: VanarSena can easily spin up multiple concurrent instances in a cloud infrastructure to accelerate fault-finding.

**Instrumentation:** The instrumenter runs over the app binary; it adds five modules to the app as shown in Figure 3-9. At run-time, these modules generate information needed for UI Automator and the Fault Inducer (§3.4).

**UI Automator:** The UI Automator (UIA) launches and navigates the instrumented app in the emulator. It emulates user interactions such as clicking buttons, filling textboxes, and swiping. It incorporates techniques to ensure both coverage and speed (§3.5).

**Fault Inducer:** During emulated execution, the Fault Inducer (FI) systematically induces different faults at appropriate points during execution (§3.6).
Monkeys

Monkey Manager

Instrumented App

Instrumenter

Submit app

App

Monkeys

Spawn

App, Config

Analysis

Developer Feedback

Developer

Figure 3-8: VanarSena Architecture.

Phone Emulator

UI Scraper
Hit Test Monitor
Transaction Tracker

Instrumented App

API Interceptors
Crash Logger

UI events, Hit Test

UI state

Handlers invoked
Processing state

Callbacks

Fault Inducer

Crash Logs

UI Automator

Config

Figure 3-9: Monkey design.
3.4 Instrumentation

We use the binary instrumentation framework described in Chapter 2 to rewrite an app binary. The instrumentation is designed for apps written using the Silverlight framework [71]. Silverlight is used by a vast majority of apps in the Windows Phone app store.

In VanarSena, the instrumentation injects five modules into the app that provides the information needed for the UI Automator and the Fault Inducer, as shown in Figure 3-9. The modules communicate with the UI Automator and the Fault Inducer via local sockets.

UI Scraper: In Silverlight, an app page is represented as a DOM tree of UI elements. The UI Scraper, when invoked, serializes the current UI page and sends it to the UIA. For each UI element, it sends the element's type, location and whether it is visible on the current screen. The UI Automator can invoke the UI Scraper on demand to inspect the current UI.

Hit Test Monitor: We instrument every event handler in the app with a Hit Test Monitor. The Hit Test Monitor helps the UIA to decide which controls to interact with. We describe hit testing in detail in §3.5.1.

Transaction Tracker: The transaction tracker provides the ProcessingCompleted event used by the UIA to decide when to interact next. We describe transaction tracking in detail in §3.5.2.

API Interceptors: The instrumenter rewrites the app to intercept certain API calls to proxy through the Fault Inducer. We describe API interceptors and Fault Inducer in detail in §3.6.

Crash Logger: To identify that an app has crashed, we rewrite the app to subscribe for the unhandled exception handler [31]. The unhandled exception handler is invoked just before the app crashes with an exception that is not handled by the developer. When the handler is invoked, we log the exception and the stack trace associated with it.

3.5 UI Automator

As the UIA navigates through the app, it needs to make two key decisions: what UI control to interact with next, and how long to wait before picking the next control. In addition, because of the design of each monkey instance, VanarSena adopts a "many randomized concurrent monkeys" approach, which we discuss below.

To pick the next control to interact with, the UIA asks the UI Scraper module (Figure 3-9) for a list of visible controls on the current page (controls may be overlaid atop each other).

In one design, the UIA can systematically explore the app by picking a control that it has not interacted with so far, and emulating pressing the back button to go back to the previous page if all controls on a page have been interacted with. If the app crashes, VanarSena generates a crash report, and the monkey terminates.

Such a simple but systematic exploration has three problems that make it unattractive. First, multiple controls often lead to the same next page. For example, clicking on any of the business categories on page 1 in Figure 3-6 leads to the Business page (page 2), a situation...
void btnFetch_Click(object sender,EventArgs e) {
    if (HitTestFlag == true) {
        HitTest.MethodInvoked(12, sender, e);
        return;
    }

    // Original Code
}

Figure 3-10: Event Handlers are instrumented to enable Hit Testing. Handler’s unique id is 12.

represented by the single edge between the pages in Figure 3-7. We can accelerate testing in this case by invoking only one of these “equivalent” controls, although it is possible that some of these may lead to failures and not others (a situation mitigated by using multiple independent monkeys).

Second, some controls do not have any event handlers attached to them. For example, the title of the page may be a text-box control that has no event handlers attached to it. UIA should not waste time interacting with such controls, because it will run no app code.

Last but not least, a systematic exploration can lead to dead ends. Imagine an app with two buttons on a page. Suppose that the app always crashes when the first button is pressed. If we use systematic exploration, the app would crash after the first button is pressed. To explore the rest of the app, the monkey manager would have to restart the app, and ensure that the UIA does not click the first button again. Maintaining such state across app invocations is complicated and makes the system more complex for many reasons, prominent among which is the reality that the app may not even display the same set of controls on every run.

We address the first two issues using a novel technique we call hit testing (§3.5.1), and the third by running multiple independent random monkeys concurrently (§3.5.3).

3.5.1 Hit Testing

Hit testing works as follows. The instrumentation framework instruments all UI event handlers in an app with a hit test monitor. It also assigns each event handler a unique ID. Figure 3-10 shows an example. When hit testing is enabled, interacting with a control will invoke the associated event handler, but the handler will simply return after informing the UIA about the invocation, without executing the event handler code.

On each new page, UIA sets the HitTestFlag and interacts with all controls on the page, one after the other. At the end of the test, the UIA can determine which controls lead to distinct event handlers. UIA can test a typical page within a few hundred milliseconds.

The arrows and the associated numbers in Figure 3-6 shows the result of hit tests on pages. For example, clicking any item on the categories page leads to the same event handler, while clicking on the word “categories” on that page does not invoke any event handler (gray arrow). In fact, the controls on the page lead to just three unique event handlers: clicking on one of the categories leads to event handler 1, clicking on settings leads to handler 2 and swiping on the page leads to handler 3. Note also that several controls on page 1 have no
event handlers attached them (gray arrows). By using hit testing, the monkey can focus only on controls that have event handlers associated with them. And from different controls associated with the same event handler, it needs to pick only one⁴, thereby significantly reducing the testing time. In §3.7.2, we will evaluate the impact of hit testing.

We stress that the binding of event handlers to controls can be dynamic, (i.e. it can be changed by the app at run time). Thus static analysis is not sufficient to determine which event handler will be triggered by a given control. This issue has also been raised in [58].

3.5.2 When to interact next?

Emulating an “open loop” or impatient user is straightforward because the monkey simply needs to invoke event handlers independent of whether the current page has properly been processed and rendered, but emulating a real, patient user who looks at the rendered page and then interacts with it is trickier. Both types of interactions are important to test. The problem with emulating a patient user is that it is not obvious when a page has been completely processed and rendered on screen. Mobile applications exhibit significant variability in the time they take to complete rendering: we show in §3.7 (Figure 3-21) that this time could vary between a few hundred milliseconds to several seconds. Waiting for the longest possible timeout using empirical data would slow the monkey down to unacceptable levels.

Fortunately, our ability to track user transactions provides a natural solution to the problem. We can generate a signal that indicates that processing of the user interaction is complete. (Unlike web pages, app pages do not have a well-defined page-loaded event [80] because app execution can be highly asynchronous. So binary instrumentation is particularly effective here.)

The processing of a user interaction is complete when all the threads and asynchronous calls triggered by the interaction is complete. In other words, the processing is complete when the user transaction ends. Figure 3-11 shows the user transaction for the interaction with "Bars" in Figure 3-6. The processing is complete after the UI update thread.

To find processing completion, we instrument the app to do transaction tracking as described in §2.3. We monitor the transaction at runtime and generate a

---

⁴In other words, we assume that two controls that lead to same event handler are equivalent and invokes similar transactions. See §3.5.3 for a caveat.
ProcessingCompleted event when all the processing (synchronous and asynchronous) associated with an interaction is complete. At runtime, we keep track of the outstanding edges in the transaction graph. When there are no more outstanding edges, we generate a ProcessingCompleted event. During TrackUpcallStart, we add an outstanding processing edge identified by the upcallMatchId. During TrackUpcallEnd, we remove the edge. Similarly, we add an outstanding edge during TrackAsyncCall and remove it during TrackAsyncCallback. The async call and callback is matched using asyncMatchId. We also correctly handle thread waits, sleeps, and timers.

3.5.3 Randomized Concurrent Monkeys

VanarSena uses many simple monkeys operating independently and at random, rather than build a single more complicated and stateful monkey.

Each monkey picks a control at random that would activate an event handler that it has not interacted with in past. For example, suppose the monkey is on page 1 of Figure 3-6, and it has already clicked on settings previously, then it would choose to either swipe (handler 3), or click one of the businesses at random (handler 1).

If no such control is found, the monkey clicks on the back button to travel to the parent page. For example, when on page 3 of Figure 3-6, the monkey has only one choice (handler 6). If it finds itself back on this page after having interacted with one of the controls, it will click the back button to navigate back to page 2. Pressing the back button in page 1 will quit the app.

Because an app can have loops in its UI structure (e.g. a “Home” button deep inside the app to navigate back to the first page), running the monkey once may not fully explore the app. To mitigate this, we run several monkeys concurrently. These monkeys do not share state, and make independent choices.

Running multiple, randomized monkeys in parallel has two advantages over a single complicated monkey. First, it overcomes the problem of deterministic crashes. Second, it can improve coverage. Note that we assumed that when two controls lead to the same event handler, they are equivalent. While this assumption generally holds, it is not a fact. One can design an app where all button clicks are handled by a single event handler, which takes different actions depending on the button’s name. Random selection of controls ensures that different monkeys would pick different controls tied to the same event handler, increasing coverage for apps that use this coding pattern.

Putting it all together: Figure 3-12 shows the overall flow of the UI automator. The experimental results in §3.7.2 demonstrate the effectiveness of hit testing and randomization, and underscore the need for ProcessingCompleted event.

3.6 Inducing Faults

The Fault Inducer (FI) is built as an extensible module in which various fault inducing modules (FIM) can be plugged in. The monkey manager configures each monkey to turn on one or more FIMs.

The FIMs are triggered by the instrumentation added to the app. The binary instrumentation rewrites the app code to intercept calls to specific APIs to proxy them through the appropriate FIM. Figure 3-13 shows an example. When the call to the HTTP API is
Figure 3-12: UI automator flow.

*Original code*

```csharp
void fetch(string url) {
    WebRequest.GetResponse(url, callback);
}
```

*Rewritten code*

```csharp
void fetch(string url) {
    WebRequestIntercept.GetResponse(url, callback);
}
```

class WebRequestIntercept {
    void GetResponse(string url, delegate callback) {
        if (MonkeyConfig.InducingResponseFaults)
            ResponseFaultInducer.Proxy(url, callback);
        if (MonkeyConfig.InducingNetworkFaults)
            NetworkFaultInducer.RaiseNetworkEvent();
    }
}

Figure 3-13: Intercepting web API to proxy through web response FIM and informing network FIM about the impending network transfer.
made at run-time, it can be proxied through the FIM that mimics web errors. The FIM may return an HTTP failure, garble the response, and so forth.

We built FIMs that help uncover some of the prominent crash buckets in Table 3.1. The first three intercept API calls and return values that apps may overlook, while the others model unexpected user behavior.

1) **Web errors:** When an app makes a HTTP call, the FIM intercepts the calls and returns HTTP error codes such as 404 (Not Found) or 502 (Bad Gateway, or unable to connect). These can trigger WebExceptions. The module can also intercept the reply and garble it to trigger parsing errors. Parsing errors are particularly important for apps that obtain data from third-party sites. We use Fiddler [3] to intercept and manipulate web requests.

2) **Poor Network conditions:** Brief disconnections and poor network conditions can trigger a variety of network errors, leading to WebExceptions. To emulate these network conditions, we instrument the app to raise an event to the FI just before an impending network transfer. The FIM can then emulate different network conditions such as brief disconnection, slow network rate, or long latency. We use a DummyNet-like tool [76] to simulate these conditions.

3) **Sensor errors:** We introduce sensor faults by returning null values and extreme values for sensors such as GPS and accelerometers.

4) **Invalid text entry:** A number of apps do not validate user inputs before parsing them. To induce these faults, the UIA and the FI work together. The UI Scraper generates an event to the FI when it encounters a textbox. The FIM then informs the UIA to either leave the textbox empty, or fill it with text, numbers, or special symbols.

5) **Impatient user:** In §3.5.2, we described how the UIA emulates a patient user by waiting for the ProcessingCompleted event. However, real users are often impatient, and may interact with the app again before processing of the previous interaction is complete. For example, in Figure 3-6, an impatient user may click on “Bars” on page 1, decide that the processing is taking too long, and click on the back button to try and exit the app. Such behavior may trigger race conditions in the app code. Table 3.1 shows that it is the root cause of many crashes. To emulate an impatient user, the transaction tracker in the app raises an event to the FI when a transaction starts, i.e., just after the UIA interacted with a control. To emulate an impatient user, the FIM then instructs the UIA to immediately interact with another specific UI control, without waiting for ProcessingCompleted event. We emulate three distinct impatient user behaviors—clicking on the same control again, clicking on another control on the page, and clicking on the back button.

It is important to be careful about when faults are induced. When a FIM is first turned on, it does not induce a fault on every intercept or event, because it can result in poor coverage. For example, consider testing the AroundMe app (Figure 3-6) for web errors. If the FIM returns 404 for every request, the app will never populate the list of businesses on page 2, and the monkey will never reach page 3 and 4 of the app. Hence, a FIM usually attempts to induce each fault with some small probability. Because VanarSena uses multiple concurrent monkeys, this approach works in practice.

During app testing, VanarSena induces only one fault at a time: each one instance of the monkey runs with just one FIM turned on. This approach helps us pinpoint the fault that is responsible for the crash. The monkey manager runs multiple monkeys concurrently with different FIMs turned on.
3.7 Evaluation

We evaluate VanarSena along two broad themes. First, we demonstrate the usefulness of the system by describing the crashes VanarSena found on 3,000 apps from the Windows Phone Store. Then, we evaluate the optimizations and heuristics described in §3.5.

To test the system, we selected apps as follows. We bucketized all apps that were in the Windows Phone app store in the first week of April 2013 into 6 groups, according to their rating (no rating, rating ≤ 1, ⋯, rating ≤ 5). We randomly selected 500 apps from each bucket. This process gives us a representative set of 3,000 apps to test VanarSena with.

We found that 15% of these apps had a textbox on the first page. These might have required user login information, but we did not create such accounts for the apps we evaluated. So it is possible (indeed, expected) that for some apps, we didn’t test much more than whether there were bugs on the sign-in screen. Despite this restriction, we report many bugs, suggesting that most (but not all) apps were tested reasonably thoroughly. In practice, we expect the developer to supply app-specific inputs such as sign-in information.

3.7.1 Crashes

We ran 10 concurrent monkeys per run, where each run tests one of the eight fault induction modules from Table 3.3, as well as one run with no fault induction. Thus, there were 9 different runs for each app, 90 monkeys in all. In these tests, the UIA emulated a patient user, except when the “impatient user” FIM was turned on.

We ran the tests on 12 machines, set up to both emulate Windows Phone 7 and Windows Phone 8 in different tests. Overall, testing 3,000 apps with 270,000 distinct monkey runs took 4,500 machine hours with each app tested for 1.5 hours on average. At current Azure pricing, the cost of testing one app is roughly 25 cents, which is small enough for nightly app tests to be done. The process emulated over 2.5 million interactions, covering over 400,000 pages.

Key Results

Overall, VanarSena flagged 2969 unique crashes in 1108 apps. Figure 3-14 shows that it found one or two crashes in 60% of the apps. Some apps had many more crashes—one had 17.

Note that these crashes were found in apps that are already in the marketplace; these are not “pre-release” apps. VanarSena found crashes in apps that have already (presumably) undergone some degree of testing by the developer.

Table 3.2 bucketizes crashed apps according to their ratings rounded to nearest integer values. Note that we have 500 total apps in each rating bucket. We see that VanarSena discovered crashes in all rating buckets. For example, 350 of the no-rating 500 apps crashed during our testing. This represents 31% of total (1108) apps that crashed. We see that the crash data in WPER for these 3000 apps has a similar rating distribution except for the 'no-rating' bucket. For this bucket, WPER sees fewer crashes than VanarSena most likely because these apps do not have enough users (hence no ratings).

---

5 The uniqueness of the crash is determined by the exception type and stack trace. If the app crashes twice in exactly the same place, we count it only once.
Figure 3-14: Crashes per app

<table>
<thead>
<tr>
<th>Rating value</th>
<th>VanarSena</th>
<th>WPER</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>350 (32%)</td>
<td>21%</td>
</tr>
<tr>
<td>1</td>
<td>127 (11%)</td>
<td>13%</td>
</tr>
<tr>
<td>2</td>
<td>146 (13%)</td>
<td>16%</td>
</tr>
<tr>
<td>3</td>
<td>194 (18%)</td>
<td>15%</td>
</tr>
<tr>
<td>4</td>
<td>185 (17%)</td>
<td>22%</td>
</tr>
<tr>
<td>5</td>
<td>106 (10%)</td>
<td>13%</td>
</tr>
</tbody>
</table>

Table 3.2: Number of crashed apps for various ratings
Comparison Against the WPER Database

It is tempting to directly compare the crashes we found with the crash reports for the same apps in the WPER database discussed in §3.1. Direct comparison, however, is not possible because both the apps and the phone OS have undergone revisions since the WPER data was collected. But we can compare some broader metrics.

VanarSena found 1,227 crashes not in the WPER database. We speculate that this is due to two key reasons. First, the database covers a period of one year. Apps that were added to the marketplace towards the end of the period may not have been run sufficiently often by users. Also, apps that are unpopular (usually poorly rated), do not get run very often in the wild, and hence do not encounter all conditions that may cause them to crash. To validate this hypothesis, we examined metadata of the apps in Windows Phone Store. The app store provides information such as rating counts and average rating values of apps, but not their actual downloads or usage counts. However, previous works have pointed out that rating count and download count of apps are strongly correlated and hence a high rating count is a strong indication of a high download count [22]. We found that apps for which VanarSena found these 1,227 crashes have, on average, 3x fewer reviews (and hence likely fewer downloads) and 10% worse rating than remaining of the apps we used.

The crashes found by VanarSena cover 16 out of 20 top crash buckets (exception name plus crash method) in WPER, and 19 of the top 20 exceptions. VanarSena does not report any OutOfMemoryException because of the following reason. To collect crashes, VanarSena instruments the unhandled exception handler inside the app. Out of memory is a fatal exception that crashes the app without calling the exception handler. WPER collects crash data at the system level instead of the app level where OutOfMemoryException is logged.

Figure 3-15 shows another way to compare VanarSena crash data and WPER. For this graph, we consider the subset of WPER crashes that belong to the crash buckets and the apps for which VanarSena found at least one crash. For each bucket, we take the apps that appear in WPER, and compute what fraction of these apps are also crashed by VanarSena. We call this fraction bucket coverage. Figure 3-15 shows that for 40% of the buckets, VanarSena crashed all the apps reported in WPER, which is a significant result suggesting good coverage.

Analysis

Even “no FIM” detects failures. Table 3.3 shows the breakdown of crashes found by VanarSena. The first row shows that even without turning any FIM on, VanarSena discovered 506 unique crashes in 429 apps (some apps crashed multiple times with distinct stack traces; also, the number of apps in this table exceeds 1108 for this reason). The main conclusion from this row is that merely exploring the app thoroughly can uncover faults. A typical exception observed for crashes in this category is the NullReferenceException. The table also shows that 239 of these 506 crashes (205 apps) were not in the WPER database.

We now consider the crashes induced by individual FIMs. To isolate the crashes caused by a FIM, we take a conservative approach. If the signature of the crash (stack trace) is also found in the crashes included in the first row (i.e., no FIM), we do not count the crash. We also manually verified a large sample of crashes to ensure that they were actually being caused by the FIM used.

Most failures are found by one or two FIMs, but some apps benefit from more FIMs. Figure 3-16 shows the number of apps that crashed as a function of the number
Figure 3-15: Coverage of crash buckets in WPER data

<table>
<thead>
<tr>
<th>FIM</th>
<th>Crashes (Apps)</th>
<th>Example crash buckets</th>
<th>Not in WPER</th>
</tr>
</thead>
<tbody>
<tr>
<td>No FIM</td>
<td>506 (429)</td>
<td>NullReferenceException, InvokeEventHandler</td>
<td>239 (205)</td>
</tr>
<tr>
<td>Text Input</td>
<td>215 (191)</td>
<td>FormatException, Int32.Parse</td>
<td>78 (68)</td>
</tr>
<tr>
<td>Impatient User</td>
<td>384 (323)</td>
<td>InvalidOperationException, Navigation.GoBack</td>
<td>102 (89)</td>
</tr>
<tr>
<td>HTTP 404</td>
<td>637 (516)</td>
<td>WebException, Browser.BeginOnUI</td>
<td>320 (294)</td>
</tr>
<tr>
<td>HTTP 502</td>
<td>339 (253)</td>
<td>EndpointNotFoundException, Browser.BeginOnUI</td>
<td>164 (142)</td>
</tr>
<tr>
<td>HTTP Bad Data</td>
<td>768 (398)</td>
<td>XmlException, ParseElement</td>
<td>274 (216)</td>
</tr>
<tr>
<td>Network Poor</td>
<td>93 (76)</td>
<td>NotSupportedException, WebClient.ClearWebClientState</td>
<td>40 (34)</td>
</tr>
<tr>
<td>GPS</td>
<td>21 (19)</td>
<td>ArgumentOutOfRangeException, GeoCoordinate..ctor</td>
<td>9 (9)</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>6 (6)</td>
<td>FormatException, Double.Parse</td>
<td>1 (1)</td>
</tr>
</tbody>
</table>

Table 3.3: Crashes found by VanarSena.
of FIMs that induced the crashes. For example, 235 apps required no FIM to crash them at all\(^6\). Most app crashes are found with less than three FIMs, but complex apps fail for multiple reasons (FIMs). Several apps don’t use text boxes, networking, or sensors, making those FIMs irrelevant, but for apps that use these facilities, the diversity of FIMs is useful. The tail of this chart is as noteworthy as the rest of the distribution.

Many apps do not check the validity of the strings entered in textboxes. We found that 191 apps crashed in 215 places due to this error. The most common exception was `FormatException`. We also found web exceptions that resulted when invalid input was proxied from the cloud service backing the app.

Emulating an impatient user uncovers several interesting crashes. Analysis of stack traces and binaries of these apps showed that the crashes fall in three broad categories. First, a number of apps violate the guidelines imposed by the Windows Phone framework regarding handling of simultaneous page navigation commands. These crashes should be fixed by following suggested programming practices [1]. Second, a number of apps fail to use proper locking in event handlers to avoid multiple simultaneous accesses to resources such as the phone camera and certain storage APIs. Finally, several apps had app-specific race conditions that were triggered by the impatient behavior.

Several apps incorrectly assume a reliable server or network. Some developers evidently assume that cloud servers and networks are reliable, and thus do not handle HTTP errors correctly. VanarSena crashed 516 apps in 637 unique places by intercepting web calls, and returning the common “404” error code. The error code representing Bad Gateway (“502”) crashed 253 apps.

Some apps are too trusting of data returned from servers. They do not account for the possibility of receiving corrupted or malformed data. Most of the crashes in this category were due to XML and JSON parsing errors. These issues are worth addressing also because of potential security concerns.

\(^6\text{This number is less than 429 (row 1 of Table 3.3), because some of those 429 apps crashed with other FIMs as well. Unlike Table 3.3, apps in Figure 3-16 add up to 1108.}\)
Some apps do not correctly handle poor network connectivity. In many cases, the request times out and generates a web exception which apps do not handle. We also found a few interesting cases of other exceptions, including a NullReferenceException, where an app waited for a fixed amount of time to receive data from a server. When network conditions were poor, the data did not arrive during the specified time. Instead of handling this possibility, the app tried to read the non-existent data.

A handful of apps do not handle sensor failures or errors. When we returned a NaN for the GPS coordinates, which indicates that the GPS is not switched on, some apps crashed with ArgumentOutOfRangeException. We also found a timing-related failure in an app where it expected to get a GPS lock within a certain amount of time, failing when that did not happen.

API compatibility across OS versions caused crashes. For example, in the latest Windows Phone OS (WP8), the behavior of several APIs has changed [2]. WP8 no longer supports the FM radio feature and developers were advised to check the OS version before using this feature. Similar changes have been made to camera and GPS APIs. To test whether the apps we selected are susceptible to API changes, we ran them with the emulator emulating WP8. The UIA emulated patient user, and no FIMs were turned on. We found that 8 apps crashed with an RadioDisabledException, while the camera APIs crashed two apps. In total, we found about 221 crashes from 212 apps due to API compatibility issues.

3.7.2 Monkey Techniques

We now evaluate the heuristics and optimizations discussed in §3.5. Unless specified otherwise, the results in this section use the same 3000 apps as before. The apps were run 10 times, with no FIM, and the UIA emulated a patient user.

Coverage

We measure coverage in terms of pages and user transactions. We desire that the monkey should cover as much of the app as possible. However, there is no easy way to determine how many unique pages or user transactions the app contains. Any static analysis may undercount the pages and controls, since some apps generate content dynamically. Static analysis may also overestimate their numbers, since apps often include 3rd party libraries that include a lot of pages and controls, only a few of which are accessible to the user at run-time.

Thus, we rely on human calibration to thoroughly explore a small number of apps and compare it to monkey's coverage. We randomly picked 35 apps and recruited 3 users to manually explore the app. They were specifically asked to click on possible controls and trigger as many unique transactions as possible. We instrumented the apps to log the pages visited and the transactions invoked. Then, we ran the app through our system, with the configuration described earlier.

In 26 out of 35 apps, the monkey covered 100% of pages and more than 90% of all transactions. In five of the remaining nine apps, the monkey covered 75% of the pages. In four apps, the monkey was hampered by the need for app-specific input such as login/passwords and did not progress far. Although this study is small, it gives us confidence that the monkey is able to explore the vast majority of apps thoroughly.

---

7Note that this data is not included in any earlier discussion (e.g. Table 3.3) since we used Windows 7 emulator for all other data.
Benefits of Hit Testing

Hit testing accelerates testing by avoiding interacting with non-invokable controls. Among invokable controls, hit testing allows the monkey to interact with only those that lead to unique event handlers.

To evaluate the usefulness of hit testing, we turned off randomization in the UIA, and ran the monkey with and without hit testing once for each app. When running without hit testing, we assume that every control leads to a unique event handler, so the monkey interacts with every control on the page.

Figure 3-17 shows the ratio of invokable controls and unique event handlers to the total controls in each app. We found that in over half the apps, less than 33% of the total controls in the app were invokable, and only 18% lead to unique event handlers.

Figure 3-18 shows the time to run apps with and without hit testing. The 90th percentile of the time to run the app once with no fault induction was 365 seconds without hit testing, and only 197 seconds with hit testing. The tail was even worse: for one particular app, a single run took 782 seconds without hit testing, while hit testing reduced the time to just 38 seconds, a 95% reduction.
Figure 3-19: Fraction of pages covered with and without hit testing.

Figure 3-20: Fraction of transactions covered with and without hit testing.
At the same time, we found that hit testing had minimal impact on app coverage (Figure 3-19 and Figure 3-20). In 95.7% of the apps, there was no difference in page coverage with and without hit testing, and for 90% of the apps, there was no difference in transaction coverage either. For the apps with less than 100% coverage, the median page and transaction coverage was over 80%.

**Importance of the ProcessingCompleted Event**

When emulating a patient user, the UIA waits for the ProcessingCompleted event to fire before interacting with the next control. Without such an event, we would need to use a fixed timeout. We now show that using such a fixed timeout is not feasible.

Figure 3-21 shows distribution of the processing time for transactions in the 3000 apps. Recall (Figure 3-11) that this includes the time taken to complete the user transaction. For this figure, we separate the transactions that involved network calls and those that did not. We also ran the apps while the FIM emulated typical 3G network speeds. This FIM affects only the duration of transactions that involve networking, and the graph shows this duration as well.

The graph shows that processing times of the transactions vary widely, from a few milliseconds to over 10 seconds. Thus, with a small static timeout, we may end up unwittingly emulating an impatient user for many transactions. Worse yet, we may miss many UI controls that are populated only after the transaction is complete. On the other hand, with a large timeout, for many transactions, the UIA would find itself waiting unnecessarily. For example, a static timeout of 4 seconds covers 90% of the normal networking transactions, but is unnecessarily long for non-networking transactions. On the other hand, this value covers only 60% of the transactions when emulating a 3G network.

This result demonstrates that using the ProcessingCompleted event allows VanarSena to maximize coverage while minimizing processing time.

**Multiple Concurrent Monkeys are Useful**

Figure 3-22 shows the CDF of the fraction of pages covered with 1, 5, and 9 monkeys compared to the pages covered with 10 monkeys. The y-axis is on a log scale. Although 85% of apps need only one monkey for 100% coverage, the tail is large. For about 1% of
Figure 3-22: Fraction of pages covered by runs compared to pages covered by 10 runs.

Figure 3-23: Fraction of transactions covered by runs compared to transactions covered by 10 runs.
the apps, new pages are discovered even by the 9th monkey. Similarly, Figure 3-23 shows that for 5% of the apps, VanarSena continues to discover new transactions even in the 9th monkey.

We did an additional experiment to demonstrate the value of multiple concurrent runs. Recall that we ran each app through each FIM 10 times. To demonstrate that it is possible to uncover more bugs if we run longer, we selected 12 apps from our set of 3000 apps that had the most crashes in WPER system. We ran these apps 100 times through each FIM. By doing so, we uncovered 86 new unique crashes among these apps (4 to 18 in each) in addition to the 60 crashes that we had discovered with the original 10 runs.

3.8 Discussion and Limitations

Why not instrument the emulator? VanarSena could have been implemented by modifying the emulator to induce faults. As a significant practical matter, however, modifying the large and complex emulator code would have required substantially more development effort than our architecture. Moreover, it would require the fault detection software to be adapted to the emulator evolving.

Why cloud deployment? We envision VanarSena as a cloud service for a couple of reasons. First, the cloud offers elastic resources – i.e. a large number of emulators can be deployed on demand. Second, a cloud-based deployment also makes for easier updates. We can update the monkey in a variety of ways – e.g. by adding more FIMs based on crash reports from the field, or by using (as yet undiscovered) techniques for improving testing speed or resource consumption. That said, it is easy to envision non-cloud deployment models as well. For example, VanarSena can work by spawning multiple VMs on developer’s desktop (resources permitting), or on a network of local machines. These other deployment scenarios have their own advantages and disadvantages. The fact that monkeys run independently of each other allows for many deployment and pricing models.

Number of monkeys per app: For results reported in this chapter, we ran a fixed number of monkeys for every app. However, we also found that running more monkeys often uncovers more crashes (although we expect that the the returns will be diminishing). We also believe that the number of monkeys needed to test an app comprehensively for each fault depends on the complexity of the app. Static analysis of the app should be able to provide some guidance in this matter. We leave this as part of our future work.

Providing inputs in a specific order: Some apps do not make progress until certain inputs are provided in a specific order. For example, log-in button should be clicked only after filling username and password. In VanarSena, we expect developers to provide inputs to textboxes. When the UI Automator discovers textboxes in a page, it fills the textboxes first (with the developer provided input or a random value) before hit testing and clicking controls. This helps the UIA to quickly go past login screens. For all other types of inputs, the UIA picks the controls randomly. If an app requires a specific order of interaction, it might be worthwhile to get hints from the developer to save testing time. We are currently exploring how developers can provide such hints easily.
**Testing Games:** Many games require complex, free-form gestures. Thus, trace replay [37] may be a more appropriate testing strategy on game apps, than randomized monkey actions. Our monkey can easily support trace replay, although collection and validation of such traces is a challenging problem that we plan to address in future. We also note that we cannot test certain other kinds of apps with the current version of VanarSena. Some apps launch other apps (e.g. web browser) and terminate. Testing such apps requires keeping careful track of different app contexts – something which we have not yet implemented. This, however, is an engineering challenge only - not a fundamental one.

**Overhead:** On average, our instrumentation increases the runtime of transactions by 0.02%. This small overhead is unlikely to affect the behavior of the app.

**False Positives:** The binary instrumentation may itself be buggy, causing “false positive” crashes. We cannot prove that we do not induce such false positives, but careful manual analysis of crash traces shows that none of the crashes occurred in the code VanarSena added.

**Combination of fault inducers:** We evaluated apps by injecting one fault at a time to focus on individual faults. In reality, multiple faults may happen at the same time. We plan to investigate this in future.

**Improving the SDK:** Some of the bugs we have uncovered should be fixed in the platform, instead of in the app. For example, crashes due to violation of simultaneous page navigation could be avoided by redesigning the API.

### 3.9 Related work

At a high level, VanarSena consists of two components: (1) dynamic analysis with a monkey, and (2) fault injection for app testing. Below we discuss how VanarSena compares with prior works in these two aspects.

**Static and dynamic analysis of mobile apps.** Several prior works have statically analyzed app binaries to uncover energy bugs [69, 79], performance problems [54], app plagiarism [24], security problems [30, 39], and privacy leaks [58, 35]. Static analysis is not suitable for our goal of uncovering runtime faults of apps since it cannot capture runtime issues such as poor network condition and corrupted or unexpected responses from cloud services. Several recent works have proposed using a monkey to automatically execute mobile apps for analysis of app’s runtime properties. AppsPlayground [74] runs apps in the Android emulator on top of a modified Android software stack (TaintDroid [28]) in order to track information flow and privacy leaks. Authors evaluate the tool with an impressive 3,968 apps. Recently A3E [14] and Orbit [85] use combinations of static and dynamic analysis to automatically generate test cases to reach various activities of an app. AMC [55] uses a dynamic analysis to check accessibility properties of vehicular apps. It has a UI Automator similar to VanarSena, but unlike our system, it clicks on every control in a given page and waits for a static timeout of 10 seconds before making the next interaction. With hit testing and processing completed event, we believe that VanarSena’s UI Automator would be much faster than AMC’s. Eprof [67] uses dynamic analysis (without a monkey) for fine-grained
energy accounting. ProtectMyPrivacy [6] uses the crowd to analyze app privacy settings and to automatically recommend app-specific privacy recommendations. All these works differ by their end goals and specific optimizations. Similarly, VanarSena differ from them in its end goal of uncovering runtime faults of apps and its novel monkey optimization techniques: hit testing and accurate processing completed event. The optimizations are general and can be used for other systems as well. We cannot directly compare the performance of our monkey with the other systems since all of them are for Android apps.

VanarSena uses AppInsight [75] to instrument Windows Phone app binaries. For Android apps, one could use similar frameworks such as SIF [44] and RetroSkeleton [25].

Mobile app testing with a monkey. Mobile app testing poses different challenges than traditional “enterprise” software, motivating researchers to develop mobile-specific solutions. Researchers have used Android Monkey [38] for automated fuzz testing [11, 12, 34, 45, 62]. Similar UI automation tools exist for other platforms. VanarSena differs from these tools is two major ways. First, the Android Monkey generates only UI events, and not the richer set of faults that VanarSena induces. Second, it does not optimize for coverage or speed like VanarSena. One can provide an automation script to the Android Monkey to guide its execution paths, but this approach is not scalable when exploring a large number of distinct execution paths.

Closest to our work is DynoDroid [59] that, like VanarSena, addresses the above problems, but with a different approach: it modifies the Android framework and involves humans at run-time to go past certain app pages (e.g., login screen). Another fundamental difference is that it manipulates only UI and system events and does not inject faults due to external factors such as bad network or event timing related to unexpected or abnormal user behavior, which are among the most common root causes in our real-world crash reports. A³E [14] and Orbit [85] use static and dynamic analysis to generate test cases to traverse different app activities, but do not inject external faults. All these systems could benefit from our crash analysis insights to decide what faults to inject.

ConVirt [56] is a related effort on mobile app testing that it explores the concept of contextual fuzzing. Under contextual fuzzing a variety of real world environmental and hardware conditions are systematically explored through both real hardware and emulation; these conditions include: user interaction, geo-locations, network conditions, and device/system configurations. To reduce the time in finding app performance problems, ConVirt implements a set of algorithms that leverage inter-app behavioral similarities. Unlike VanarSena, ConVirt takes a blackbox approach and incorporates actual hardware into the testing process.

Other software testing techniques. Software testing has a rich history, which cannot be covered in a few paragraphs. We focus only on recent work on mobile app testing, which falls into three broad categories: fuzz testing, which generates random inputs to apps; symbolic testing, which tests an app by symbolically executing it; and model-based testing. Fuzz testing is done with a monkey and is discussed above.

Symbolic execution [52, 20, 62] and its hybrid variant, concolic execution [12, 49] have found limited success in testing real-world apps due to path explosion problem and difficulty in modeling real-world execution environment with network, sensors, and cloud.

“GUI ripping” [60, 42, 11] systems and GUITAR [41] use model-based testing to mobile apps. Unlike VanarSena, it requires developers to provide a model of the app’s GUI and can only check faults due to user inputs. Applicability of these techniques has so far been very
limited (e.g., evaluated with a handful of “toy” apps only).

3.10 Chapter Summary

In this chapter, we described VanarSena, a system to identify mobile app failures before the app is deployed. The system is designed by gleaning insights from an analysis of 25 million crash reports. VanarSena adopts a “greybox” testing method, instrumenting the app binary to achieve both high coverage and speed, using hit testing and generation of ProcessingCompleted event. We found that VanarSena is effective in practice. We tested it on 3000 apps from the Windows Phone store, finding that 1108 of them had failures. VanarSena uncovered over 2969 distinct bugs in existing apps, including over 1227 that were not previously reported. Each app was tested, on average, in just 1.5 hours. Deployed as a cloud service, VanarSena can provide an automated testing framework to mobile software reliability even for amateur developers who cannot devote extensive resources to testing.
Chapter 4

Monitoring In The Wild

Testing before deployment is important, but is seldom sufficient. VanarSena can uncover common failures, but cannot find all possible issues that could potentially occur in the wild. Further, it is hard to characterize user-perceived performance and find performance problems in an emulated setting. The mobile environment is complex and varying, and a full range of usage conditions is difficult to emulate in a test setting. Network connectivity, GPS-signal quality, and phone hardware all vary widely. Some platform APIs even change their behavior depending on the battery level. Thus, in addition to testing, to improve the quality of their apps, developers must understand how the apps perform in the wild. To do this, collection of diagnostic and performance trace data from the field is essential.

Today, there is little platform support for app monitoring in the wild. Major mobile platforms, including iOS, Android, and Windows Phone, report app-crash logs to developers, but it is often difficult to identify the causes of crashes from these logs [5], and this data does not help diagnose performance problems. Analytics frameworks such as Flurry [32], and Preemptive [72] are designed to collect usage analytics (e.g., user demographics), rather than performance data. Thus, the only option left is for the developer to include custom tracing code in the app. However, writing such code is no easy task. Mobile apps are highly asynchronous. Even a simple user request triggers multiple asynchronous calls, with complex synchronization between threads. Identifying performance bottlenecks in such code requires correctly tracking causality across asynchronous boundaries. This challenging task is made even more difficult because tracing overhead must be minimized to avoid impact on app performance, and also to limit the consumption of scarce resources such as battery and network bandwidth.

In this chapter, we describe a system called AppInsight to help the app developers diagnose performance bottlenecks and failures experienced by their apps in the wild. AppInsight provides the developers with information on the critical path through their code for every user transaction. This information points the developer to the optimizations needed for improving user experience.

AppInsight automatically instruments the app using techniques described in Chapter 2 to collect performance and failure data from the wild. It carefully selects code points to instrument to minimize overhead. AppInsight analyzes the collected data and provides detailed feedback to the developer about performance bottlenecks, failures, and their root causes. Using AppInsight requires zero developer effort. We do not require app developers to write additional code, or add code annotations. AppInsight does not require any changes to the mobile OS or runtime and hence is readily deployable.
We have implemented AppInsight for the Windows Phone platform. To evaluate AppInsight, we instrumented 30 popular apps and recruited 30 users to use these apps on their personal phones for over 4 months. This deployment yielded trace data for 6,752 app sessions, totaling over 33,000 minutes of usage time. Our evaluation shows that AppInsight is lightweight — on average, it increases the run time by 0.021%, and the worst-case overhead is less than 0.5%. Despite the low overhead, the instrumentation is comprehensive enough to allow us to make several detailed observations about app performance in the wild. For example, we can automatically highlight the critical paths for the longest user transactions. We can also group similar user transactions together and correlate variability in their performance with variation in the environment. In §4.6.5, we will discuss how this feedback helped developers improve the quality of their app.

4.1 Goals

Our goal is to help developers understand the performance bottlenecks and failures experienced by their apps in the wild. We do this by providing them with critical paths for user transactions and exception paths when apps fail during a transaction. We now define these terms.

Critical path The critical path is the bottleneck path in a user transaction (§2), such that changing the length of any part of the critical path will change the user-perceived latency. Informally, the critical path starts with a user manipulation event, and ends with a UI update event. In Figure 2-3, the entire path from (0) to (8) constitutes the critical path of the transaction. The latency can be reduced either by reducing the download delay (4) or the processing delay (6). In Figure 2-5, the critical path is shown in bold. Note that activities related to the download and processing of the first web request are not on the critical path.

The critical path identifies the portions of the code that directly impacts user-perceived latency. However, the critical path may not always accurately characterize user experience. For example, a transaction may make multiple updates to the UI (one after the other), and the user may care about only one of them, or the user may interrupt a transaction to start a new one. We discuss this in §4.4.2.

While the critical path is useful for understanding performance bottlenecks, to debug app failures, we provide the developer with exception paths.

Exception path The exception path is the path from the user manipulation to the exception method, spanning asynchronous boundaries. In Figure 2-6, (0) to (8) is the exception path. The exception path points the developer to the user interaction that started the asynchronous path leading to the crash.

We now describe how we collect the trace data needed to deliver the above information to the developer, while minimizing the impact on application performance.

4.2 AppInsight Design Overview

Figure 4-1 shows the architecture of AppInsight. We provide a instrumentation tool (the instrumenter) for developers to automatically instrument the app. The developer only needs
Figure 4-1: AppInsight System Overview
to provide the instrumenter with app binaries; no other input or source code annotation is needed.

The instrumenter rewrites the app binary to monitor user transactions as described in Chapter 2. When users run the instrumented app, trace data is collected and uploaded to a server. We use the background transfer service (BTS) [19] to upload the trace data. BTS uploads the data when no foreground apps are running. It also provides a reliable transfer service in the face of network outages and losses. The trace data is analyzed and the findings are made available to the developer via a web-based interface (§4.5).

4.3 Instrumenter

The Instrumenter instruments the app to capture, with minimal overhead, the information necessary to build execution traces of user transactions and identify their critical paths and exception paths.

Section 2.3 in Chapter 2 explains our instrumentation in detail. We capture six categories of data: (i) when the user interacts with the UI; (ii) when the app code executes on various threads (iii) causality between asynchronous calls and callbacks; (iv) thread synchronization points and their causal relationship; (v) when the UI was updated; (vi) any unhandled exceptions. Apart from this, we also capture some additional data, as discussed in §4.3.1.

In ApplInsight, the Tracker methods insert trace records into a memory buffer. Each record is tagged with a timestamp. The buffer is flushed to stable storage to prevent overflow as needed. When the app exits, the buffer is scheduled for upload using BTS. Table 2.1 shows a sample trace for the code in Figure 2-10. The records are compressed before they are uploaded to the server. During instrumentation, the mapping between method Id and method names, and call Id and call signature are stored in a metadata file and uploaded to the analysis server for proving feedback to the developer.

4.3.1 Capturing Additional Information

For certain asynchronous calls such as web requests and GPS calls, we collect additional information both at the call and at the callback. For example, for web request calls, we log the URL and the network state. For GPS calls, we log the state of the GPS. The choice of the information we log is guided by our experience, and the inevitable tradeoff between completeness and overhead. Our data shows that critical paths in a user transaction often involve either network or GPS accesses. By logging a small amount of additional information at certain points, we can give more meaningful feedback to the developer.

4.4 Analysis Methodology

We analyze the traces to delineate individual user transactions, and identify critical paths and exception paths. Transactions can also be analyzed in aggregate, to highlight broader trends.

4.4.1 User transactions

We represent user transactions by directed acyclic graphs. The graph is generated from the trace data. Consider the trace in Table 2.1. It is converted to the graph in Figure 4-2.
The graph contains five types of nodes, namely: (I) User Interaction, (S) Upcall start, (E) Upcall end, (A) Async call start, and (L) Layout updated. Each node represents one trace record\(^1\) and is identified by the type and the record id. The mapping between node types I, S, E, A and L and the record types can be gleaned from Table 2.1.

The edges between nodes represent causal relationships. For example, the user interaction event I1 triggers the start of the handler S2. Similarly, the start of callback execution S7 was caused by the asynchronous call A3. We also say that an upcall start node "causes" any subsequent activity on that upcall. Hence we draw S2 → A3, as the async call was made during execution of the upcall, and S2 → E4, to represent the fact that the upcall end is triggered by upcall start.

The above graph does not show any thread synchronization events. These are represented by three types of nodes, namely: (B) Thread blocked node, (F) Semaphore fired node, and (W) Thread wakeup node. We’ll describe these nodes later.

When the app trace contains overlapping user transactions, this approach correctly separates them, and generates a graph for each.

We now discuss how we use this graphical representation to discover the critical path in a user transaction.

### 4.4.2 Critical Path

The critical path is the bottleneck path in the user transaction (§4.1). The basic algorithm for finding the critical path is simple. Consider Figure 4-2. We traverse the graph backwards, going from the last UI update (L13), to the user interaction event that signals the start of the transaction (I1), traversing each directed edge in the opposite direction. This path\(^2\), when reversed, yields the critical path: I1, S2, A3, S7, A8, S11, E12, L13. Even this simple example shows that we correctly account for time spent inside upcalls: for example, the edge (S7, E9) is not on the critical path, which means that any activity in the reqCallback

---

\(^1\)AsyncCallback records are used for bookkeeping purposes only, and are not mapped to nodes.

\(^2\)This algorithm always terminates because the transaction graph is always acyclic. Also, we are guaranteed to reach an I node from an L node, with backward traversal.
method (See Figure 2-10), after calling the dispatcher, does not affect user-perceived latency. This basic algorithm requires several refinements, as discussed below.

Multiple UI Updates: As discussed in §4.1, the transaction may update the UI multiple times. This results in multiple $L$ nodes in the transaction graph. Only the developer can accurately determine which of these updates is important. In such cases, AppInsight, by default, reports the critical path to the last $L$ node. However, using the feedback interface (§4.5), the developer can ask AppInsight to generate the critical path to any of the $L$ nodes.

Thread synchronization via signaling: The basic algorithm implicitly assumes that each node will have only one edge incident upon it. This is not the case for the graph shown in Figure 4-3, which represents the transaction shown in Figure 2-5: Node $W$, which is a thread wakeup node, has two edges incident upon it, since the thread was waiting for two semaphores to fire (the two $F$ nodes). In such cases, we compare the timestamps of the semaphore-fire records, and pick the later event. This yields the critical path shown in the figure.

Periodic timers: An app may start a periodic timer, which fires at regular intervals and performs various tasks, including UI updates. In some cases, periodic timers can also be used for thread synchronization (§4.7). We detect this pattern, and then assume each timer firing to be the start of a separate transaction. We call these transactions timer transactions, to distinguish them from user transactions. We separate these transactions from user transactions during developer feedback. We handle sensor-driven transactions in a similar manner.
4.4.3 Exception path

When the app crashes, we log the exception information including the stack trace of the thread that crashed (§2.3.6). We also have the AppInsight-generated trace until that point. We walk the stack frames until we find a frame that contains the method name of the last UpcallStart record in the AppInsight trace. The path from the start of the transaction to the Upcall start node, combined with the stack trace represents the exception path.

4.4.4 Aggregate Analysis

AppInsight helps the developer see the “big picture” by analyzing the transactions in aggregate. There are a number of ways to look at the aggregate data. Our experience shows that the developer benefits the most by using the aggregate data to uncover the root causes of performance variability, and to discover “outliers” – i.e. transactions that took abnormally long to complete compared to similar transactions.

To perform this analysis, we group together transactions with identical graphs; i.e. they have the same nodes and the same connectivity. These transactions represent the same user interaction with the app. This is a conservative grouping; the same user interaction may occasionally generate different transaction graphs, but if two transactions have the same graph, with a high probability they correspond to the same interaction.

Understanding performance variance While the transactions in a group have the same transaction graph, their critical paths and durations can differ. To identify the major sources behind this variability, we use a standard statistical technique called Analysis of Variance (ANOVA). ANOVA quantifies the amount of variance in a measure that can be attributed to individual factors that contribute to the measure. Factors include network transfer, local processing, and GPS queries which in turn can vary because of network type, device type, GPS state, user state, etc. We will discuss ANOVA analysis in more detail in §4.6.4.

Outliers AppInsight also flags outlier transactions to help developers identify performance bottlenecks. Transactions with duration greater than \((\text{mean} + (k \times \text{standard deviation}))\) in the group are marked as outliers. We use \(k = 3\) for our analysis.

4.5 Developer Feedback

The AppInsight server analyzes the collected traces using the methods described in §4.4. The developers use a web UI to access the results. Figure 4-4 shows a collage of some of the views in the UI.

For ease of navigation, the UI groups together identical transactions (§4.4.4) (a) in Figure 4-4). To allow easy mapping to the source code, groups are named by their entry event handler method. Within each group, transactions are sorted by duration and outliers are highlighted (b). Developers can select individual transactions to view their transaction graph which are shown as interactive plots (c). The plot also highlights the critical path (d). Within a critical path, we show the time spent on each component (e). The developer can thus easily identify the parts of the code that need to be optimized. Additional information, such as URLs and network type (3G or Wi-Fi) for web calls and the state of the GPS is also shown (e). We also provide variability analysis for each transaction group (f).
The UI also shows where each transaction fits within the particular app session. This view provides developers with the context in which a particular transaction occurred (e.g., at the start of a session).

The tool also reports certain common patterns within a group and across all transactions for an app. For example, it reports the most common critical paths in a transaction group, the most frequent transactions, common sequence of transactions, frequently interrupted transactions, etc. Using this information, developers can focus their efforts on optimizing the common case.

Developers can also browse through crash reports. Crashes are grouped by their exception path. For each exception, the tool reports the exception type, shows the stack trace attached to the execution graph, and highlights the exception path.

4.6 Results

We first present results from the live deployment of AppInsight, and some case studies of how AppInsight helped developers improve their app. Then, we present micro-benchmarks to quantify AppInsight’s overhead and coverage.

4.6.1 Deployment

To select the apps to evaluate AppInsight with, we asked 50 of our colleagues to list 15 apps they regularly use on their Windows Phone. From these, we picked 29 most popular free apps. We also included an app that was developed by me. The app was published several months before we started the AppInsight project, as an independent effort. We instrumented these 30 apps using AppInsight. Thirty users volunteered to run some of the instrumented apps on their personal phones. Often, they were already using many of the apps, so we simply replaced the original version with the instrumented version. All participants had their own unlimited 3G data plan.
<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total num of apps</td>
<td>30</td>
</tr>
<tr>
<td>total participants</td>
<td>30</td>
</tr>
<tr>
<td>unique hardware models</td>
<td>6</td>
</tr>
<tr>
<td>unique hardware+firmware</td>
<td>14</td>
</tr>
<tr>
<td>start date</td>
<td>03 April 2012</td>
</tr>
<tr>
<td>end date</td>
<td>15 August 2012</td>
</tr>
<tr>
<td>total num of app launches (sessions)</td>
<td>6752</td>
</tr>
<tr>
<td>total minutes in apps</td>
<td>33,060</td>
</tr>
<tr>
<td>total user transactions</td>
<td>167,286</td>
</tr>
<tr>
<td>total timer transactions</td>
<td>392,768</td>
</tr>
<tr>
<td>total sensor transactions</td>
<td>3587</td>
</tr>
</tbody>
</table>

Table 4.1: Summary statistics from our deployment

Figure 4-5: CDF of user-transaction duration.

Table 4.1 shows summary deployment statistics. Our data comes from 6 different hardware models. Over the course of deployment, we collected trace data from 6752 app sessions. There are a total of 563,641 transactions in this data. Over 69% of these are timer transactions, triggered by periodic timers (see §4.4.2). Almost all of them are due to a one-second timer used in one of the gaming apps. In the rest of the section, we focus only on the 167,286 user transactions that we discovered in this data.

Table 4.2 shows basic usage statistics for some of the apps. Note the diversity in how often users ran each app, for how long, and how many user transactions were in each session. Over 40% of the user transactions were generated by a multiplayer game app. Figure 4-5 shows the CDF of the length of user transactions (i.e., the length of their critical path). Only 15% of the transactions last more than 5 seconds. The app developer would likely want to focus his debugging and optimization efforts on these long-running transactions.
<table>
<thead>
<tr>
<th>App description</th>
<th># Users</th>
<th>Sessions</th>
<th>Avg session length (s)</th>
<th>#User trans-actions</th>
<th>#Async calls</th>
<th>Avg parallel interrupted</th>
<th>#trans overhead ms/trans</th>
<th>Perf overhead ms/s</th>
<th>Perf overhead ms/s</th>
<th>Network overhead b/trans</th>
<th>Extra data transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>News aggregator</td>
<td>22</td>
<td>604</td>
<td>88</td>
<td>17738</td>
<td>42626</td>
<td>5.53</td>
<td>1732</td>
<td>3.02</td>
<td>0.69</td>
<td>311</td>
<td>3.2%</td>
</tr>
<tr>
<td>Weather</td>
<td>25</td>
<td>533</td>
<td>31</td>
<td>4692</td>
<td>8106</td>
<td>1.92</td>
<td>541</td>
<td>0.31</td>
<td>0.09</td>
<td>162</td>
<td>2.9%</td>
</tr>
<tr>
<td>Stock information</td>
<td>17</td>
<td>460</td>
<td>32</td>
<td>4533</td>
<td>5620</td>
<td>1.00</td>
<td>486</td>
<td>0.20</td>
<td>0.06</td>
<td>91</td>
<td>8.6%</td>
</tr>
<tr>
<td>Social networking</td>
<td>22</td>
<td>1380</td>
<td>622</td>
<td>48441</td>
<td>900802</td>
<td>7.60</td>
<td>6782</td>
<td>3.48</td>
<td>0.21</td>
<td>487</td>
<td>8.0%</td>
</tr>
<tr>
<td>Multiplayer game</td>
<td>21</td>
<td>1762</td>
<td>376</td>
<td>68757</td>
<td>359006</td>
<td>2.28</td>
<td>719</td>
<td>0.18</td>
<td>0.26</td>
<td>27</td>
<td>79.0%</td>
</tr>
<tr>
<td>Transit info</td>
<td>7</td>
<td>310</td>
<td>37</td>
<td>1945</td>
<td>40448</td>
<td>4.88</td>
<td>182</td>
<td>2.96</td>
<td>0.85</td>
<td>355</td>
<td>0.9%</td>
</tr>
<tr>
<td>Group discounts</td>
<td>9</td>
<td>67</td>
<td>306</td>
<td>1197</td>
<td>3040</td>
<td>6.62</td>
<td>109</td>
<td>0.99</td>
<td>0.06</td>
<td>212</td>
<td>2.3%</td>
</tr>
<tr>
<td>Movie reviews</td>
<td>7</td>
<td>48</td>
<td>394</td>
<td>1083</td>
<td>7305</td>
<td>6.56</td>
<td>50</td>
<td>0.51</td>
<td>0.08</td>
<td>97</td>
<td>2.7%</td>
</tr>
<tr>
<td>Gas station prices</td>
<td>8</td>
<td>110</td>
<td>48</td>
<td>1434</td>
<td>2085</td>
<td>2.11</td>
<td>72</td>
<td>0.14</td>
<td>0.04</td>
<td>91</td>
<td>1.9%</td>
</tr>
<tr>
<td>Online shopping</td>
<td>14</td>
<td>43</td>
<td>512</td>
<td>1705</td>
<td>25701</td>
<td>2.74</td>
<td>349</td>
<td>0.18</td>
<td>0.06</td>
<td>24</td>
<td>4.7%</td>
</tr>
<tr>
<td>Microblogging</td>
<td>3</td>
<td>333</td>
<td>60</td>
<td>3913</td>
<td>19853</td>
<td>2.02</td>
<td>386</td>
<td>0.89</td>
<td>0.28</td>
<td>181</td>
<td>2.2%</td>
</tr>
<tr>
<td>Newspaper</td>
<td>10</td>
<td>524</td>
<td>142</td>
<td>13281</td>
<td>24571</td>
<td>4.85</td>
<td>662</td>
<td>0.33</td>
<td>0.06</td>
<td>92</td>
<td>1.2%</td>
</tr>
<tr>
<td>Ticket service</td>
<td>7</td>
<td>64</td>
<td>530</td>
<td>171</td>
<td>9533</td>
<td>3.70</td>
<td>38</td>
<td>0.06</td>
<td>0.57</td>
<td>9</td>
<td>2.9%</td>
</tr>
</tbody>
</table>

Table 4.2: Summary statistics for 13 of the 30 apps. For conciseness, we highlight a single app out of each of the major app categories. The name of the app is anonymized. Overhead data is explained in §4.6.6.
4.6.2 User Transactions and Critical Paths

In Table 4.2, we see that the average number of asynchronous calls per user transaction varies from 1.2 to 18.6 depending on the app. The average number of parallel threads per user transaction varies from 1 to 7.6. This high degree of concurrency in mobile apps is one of the key reasons why a system such as AppInsight is needed to identify the critical path in the complex graph that represents each user transaction.

Figure 4-6 offers another perspective on the complexity of user transactions and the value of AppInsight. It shows the CDF of the number of edges in a user transaction. While we have clipped the horizontal axis of this graph for clarity, there are user transactions with thousands of edges. Amidst this complexity, AppInsight helps the developers by identifying the critical path that limits the user-perceived performance. As the figure shows, the number of edges in critical paths are much fewer.

We also observe that not all edges in a critical path consume the same amount of time. Rather a few edges are responsible for most of the time taken by a transaction, as shown in Figure 4-7. This graph plots the cumulative fraction of transaction time as a function of the number of edges. We see that two edges are responsible for 82% of the transaction time. Application developers can focus on these edges to understand and alleviate the performance bottlenecks in their applications.

Investigating these time-hogging edges in critical paths, we find, expectedly, that network transfers are often to blame. In transactions that involve at least one network transfer (14.6% of total), 93% had at least one network transfer in the critical path and 35% had at least two. On an average, apps spend between 34-85% of the time in the critical path doing network transfer.

In contrast, location queries are not a major factor. In transactions that had a location query (0.03% of total), the query was in the critical path in only 19% of the cases. This occurs because most apps request for coarse location using WiFi or cell towers, without initializing the GPS device. Coarse location queries tend to be fast.
4.6.3 Exception paths

AppInsight also helps in failure diagnosis. In our deployment, we collected 111 crash logs (from 16 apps), 43 of which involved asynchronous transactions where the standard stack trace that the mobile platform gives the app developer would not have identified the full path that led up to the crash.

4.6.4 Aggregate Analysis

We analyzed the trace data from our deployment using techniques described in §4.4.4. For the data in Table 4.1, we have 6,606 transaction groups across all apps.

Understanding performance variance We first quantify the variance in transaction groups and then analyze the sources of the variance.

We find that 29% of the transaction groups contain multiple distinct critical paths. Further, even where there is a unique critical path, the dominant edge (the one that consumes most time) varies in 40% of the cases. This implies that the performance bottlenecks differ for different transactions even when the transactions correspond to the same activity.

Figure 4-8 shows the extent of performance variability we observe across transactions in a group. For each group, it plots the range (maximum - minimum) of transaction duration observed as a function of the average transaction duration. We see many activity groups with highly variable transaction duration. To show that this variability is not limited to cases with network transfers or location queries, we separately show activities that do not involve these two functions. While such activities have lower transaction duration on average, they too have highly variable performance. This variability can stem from the user’s device, system load, user state, etc.

We identify the major sources behind the variability in transaction duration using ANOVA (§4.4.4). At the highest level, there are three factors that impact transaction duration: (i) network transfer, (ii) location queries, and (iii) local processing. Each of these factors can itself vary because of network type, device type, GPS state, user state, etc. For each transaction, we split the transaction duration into these three factors depending on where time is spent on the critical path and then find the contribution of each component.

Figure 4-7: Cumulative fraction of time in the critical path as a function of number of edges.
to the variability of the transaction duration. For this analysis, we only use activity groups that have at least 10 transactions.

We focus first on activities that do not involve location queries. We find that the average contribution of network and local processing to the variability in the transaction duration was 66% and 34%. Much of the variability in transaction duration stems from the variability in network transfer time. Though, in 10% of the groups, local processing contributed to over 90% of the variability.

We further analyze those groups where network transfers were responsible for over 90% of the variability. We find that network type (3G or WiFi) plays an important role. On average, 3G-based transactions took 78% longer and had 155% more standard deviation compared to WiFi-based transactions. However, we also found groups with high variability in network-transfer time irrespective of the network type. This variation might be due to factors such as dynamic content and server delay that we do not capture.

We also analyze groups in which local processing was responsible for over 90% of the variability. We find groups where the variability can be entirely explained by the device type. For instance, in one group, transactions from Nokia Lumia 900 phones had 38% lower transaction times than those from Samsung Focus phones. One of the key differences between the two phones is that the Nokia has a 1.4 GHz processor compared to the Samsung with a 1 GHz processor. We also find transactions where the variability could be completely explained by the user herself. The duration of these transactions likely depend on user state that we do not capture.

Next, we analyze groups that have location queries in the critical path. We find that such queries contribute to the transaction duration variability in only one group. This is because, as noted above, most apps query for coarse location which is quick. In the group that queried for fine-grained location, the transaction time was highly correlated with the state of the GPS device. If it was not initialized, the query took 3–20 seconds; otherwise, it took roughly 1 ms.

**Outliers** AppInsight flags transactions that take significantly longer than other transactions in the same group (§4.4.4). Overall, we find 831 outlier transactions and 287 groups with at least one outlier. These outliers span across 11 apps. 19% of the outliers are due to large network delays (with the transaction’s network time being greater than the mean
network time in the group by more than three orders of standard deviation), 76% are due to local processing and 5% are due to both. 70% of the transaction with large network delay was on 3G. The mean transaction duration of outliers with network delay was 16.6 seconds (14.1s median), and those because of local processing delay was 10 seconds (7.4s median). From the data, we can see that, local processing also plays a major role in long transactions.

Interestingly, the factors that explain most of the variability in a transaction group can be different from those that lead to outliers. We find groups in our data where the variability was primarily due to network transfers but the outlier was due to local processing.

4.6.5 Case Studies

We now describe how AppInsight helped app developers improve their applications.

App 1

One of the apps in our deployment was developed by me (see §4.6.1). AppInsight feedback helped me improve the app in many ways. The following observations are based on 34 session traces representing 244 user transactions and 4 exception logs.

Exceptions Before being instrumented with AppInsight, the app had been on the marketplace for 1.5 years. I had occasionally received crash logs from the Windows Phone developer portal, but logs contained only the stack trace of the thread that crashed. While I knew that a routine that split a line into words was crashing, there was not enough information to diagnose the failure. When the app was instrumented with AppInsight, I received the entire exception path. This included the web call and the URL from where the line was fetched. I replayed the URL in the app in a controlled setting, and discovered that the text-parsing routines did not correctly handle certain patterns of blank lines.

UI sluggishness The aggregate analysis in AppInsight identified a user transaction with high variability in duration. The variability was attributed to local processing (time spent on thread execution). I spotted that only the user transactions at the start of user sessions experienced these abnormal latencies. I identified that certain system calls early in the app execution caused system DLLs to be loaded into memory. The time to load the DLLs was high and highly variable. Later transactions that used the same APIs did not experience high latency, as the DLLs were cached. This problem was not spotted in lab testing, since the DLLs are almost always in memory, due to continuous test runs. I redesigned the code to force-load the DLLs earlier.

Wasted computation The feedback UI pointed me to frequently interrupted transactions. I noticed that in some cases, the background threads initiated by the interrupted transaction were not being terminated, thereby wasting the battery. I modified the code to fix the problem.

Serial network operations I noticed that a common critical path consisted of web requests that were issued in a serial manner. I improved the user response time by issuing them in parallel.
App 2

AppInsight can help the developers optimize a "mature" app, that rarely experiences performance problems. For example, a popular app in our deployment has been in the marketplace for over 2 years and had gone through multiple rounds of updates. Our deployment traces had over 300 user sessions for this app, representing 1954 user transactions.

Aggregate analysis showed that 3G data latency significantly impacted certain common transactions in their app. In this case, the app developers were already aware of this problem and had considered adding caching to their app. However, they did not have good quantitative data to back up their decision. They were also impressed by the ease with which AppInsight highlighted the problem, for it had taken them a long time to pinpoint the fix. The developers are considering using AppInsight for their next release, especially to evaluate changes to the data caching policies.

App 3

We also instrumented an app that is under active development. This app was not part of our deployment – the developers tested the instrumented app in a small pilot of their own. Surprisingly, AppInsight revealed that custom instrumentation code that the developers had added was a major contributor to the poor performance of their app.

Analysis of trace data from other apps in our deployment has also shown many cases of wasteful computation, UI sluggishness, and serial network transactions in the critical path.

4.6.6 Micro-benchmarks

We now present micro-benchmarks to quantify AppInsight's overheads, and verify that AppInsight does not miss any user transactions.

Overheads

App run time  The impact of AppInsight on run time of the app is negligible. Individual logging operations simply write a small amount of data to a memory buffer, and hence are quite lightweight, as seen from Table 4.3. The buffer is flushed to disk when full\(^3\) or when the app exits. In most cases, the buffer never gets full, so flushing happens only when the app exits. The disk write happens on a background thread, and takes only a few milliseconds.

To estimate the cumulative impact of logging operations on the apps that our users ran, we multiply the number of log calls in each user transaction by overheads reported in Table 4.3. The maximum overhead per user transaction is 30ms (average 0.57ms). Since most transactions are several seconds long (see Figure 4-5), we also calculated the approximate overhead per second. The maximum overhead is 5ms (average 0.21ms) per second. We believe that this is negligible. Table 4.2 shows the average overhead per transaction and per second for different apps. The overhead is quite low. We also note that our users reported no cases of performance degradation.

Memory  AppInsight uses a 1MB memory buffer. Typical apps consume around 50MB of memory, so the memory overhead is just 2%.

\(^3\)We use a two-stage buffer to prevent data loss during flushing.
### Log Method Overhead (μs)

<table>
<thead>
<tr>
<th>Log Method</th>
<th>Overhead (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogUpcallStart</td>
<td>6</td>
</tr>
<tr>
<td>LogUpcallEnd</td>
<td>6</td>
</tr>
<tr>
<td>LogCallStart</td>
<td>6</td>
</tr>
<tr>
<td>LogCallEnd</td>
<td>6</td>
</tr>
<tr>
<td>LogCallbackStart</td>
<td>6</td>
</tr>
<tr>
<td>LogAsyncStart</td>
<td>12</td>
</tr>
<tr>
<td>LogObject</td>
<td>12</td>
</tr>
<tr>
<td>LogParameters</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 4.3: Overhead of AppInsight Logger. Averaged over 1 million runs, on a commonly used phone model.

**Network**  
AppInsight writes log records in a concise format to minimize the amount of data that must be uploaded. The median amount of trace data we upload is 3.8KB per app launch. We believe that this overhead is acceptable. We use two more metrics to further characterize the network overhead: (i) bytes per transaction and (ii) percentage of extra data transferred because of AppInsight compared to data consumed by the app. The last two columns of Table 4.2 shows these metrics for different apps. We see that the extra network overhead introduced by AppInsight is minimal for most apps. Recall that we use BTS (§4.2) to upload the data, which ensures that the upload does not interfere with the app’s own communication. BTS also provides a “Wi-Fi Only” option, which defers data upload till the phone is connected to Wi-Fi.

**Size**  
On average, the added instrumentation increased the size of the app binaries by just 1.2%.

**Battery**  
The impact of AppInsight on battery life is negligible. We measured the overhead using a hardware power meter. We ran an instrumented app and the corresponding original app 10 times each. In each run, we manually performed the same UI actions. For the original app, the average time we spent in the app was 18.7 seconds across the 10 runs, and the average power consumption was 1193 mW, with a standard deviation of 34.8. For the instrumented version, the average time spent was also 18.7 seconds, and the average power consumption was 1205 mW. This 1% increase in power consumption is well within experimental noise (the standard deviation).

**Coverage**  
AppInsight uses several heuristics (see §4.3) to reduce the amount of trace data it collects. To verify that we did not miss any user transactions because of these heuristics, we carried out a controlled experiment. First, we added extra instrumentation to the 30 apps that logs every method call as the app runs. Then, we ran these “fully instrumented” apps in a virtualized Windows Phone environment, driven by an automated UI framework similar to VanarSena, which simulates random user actions – tap screen at random places, random swiping, etc. We ran each app a 100 times, simulating between 10 and 30 user transactions each time.

---

4Some apps require non-random input at the beginning.
Upon analyzing the logs, we found that the “extra” instrumentation did not discover any new user transaction. Thus we believe that AppInsight captures necessary data to reconstruct all user transactions. We also note that the full instrumentation overhead was as much as 7,000 times higher than AppInsight instrumentation. Thus, the savings achieved by AppInsight are significant.

4.7 Discussion and Limitations

We now discuss some of the overarching issues related to AppInsight design.

Causal relationships between threads AppInsight can miss certain casual relationship between threads. First, it does not track data dependencies. For example, two threads may use a shared variable to synchronize, wherein one thread would periodically poll for data written by another thread. Currently, AppInsight uses several heuristics to identify these programming patterns, and warns the developer that the inferred critical path may be incorrect. Tracking all data dependencies requires platform support [28], which we do not have. Second, AppInsight will miss implicit causal relationships, introduced by resource contention. For example, disk I/O requests made by two threads will get serviced one after the other, introducing an implicit dependency between the two threads. Monitoring such dependencies also requires platform support. Finally, AppInsight does not track any state that a user transaction may leave behind. Thus, we miss dependencies resulting from such saved state.

Definition of user transaction and critical path The definition of user transaction and critical path in §4.1 does not address all scenarios. For example, some user interactions may involve multiple user inputs. Our current definition will break such interactions into multiple transactions. This may be incompatible with the developer’s intuition of what constitutes a transaction. In case of multiple updates to the UI, our analysis produces one critical path for each update (§4.4.2). It is up to the developer to determine which of these paths are important to investigate. Despite these limitations, results in §4.6 show that we can give useful feedback to the developer.

Privacy Any system that collects trace data from user devices risks violating the user’s privacy. To mitigate this risk, AppInsight does not store user or phone ids. Instead, we tag trace records with an anonymous hash value that is unique to that phone and that app. Since two apps running on the same phone are guaranteed to generate a different hash, it is difficult to correlate the trace data generated by different apps. This mechanism is by no means foolproof, especially since AppInsight collects data such as URLs accessed by the app. We continue to investigate this area further.

4.8 Related Work

While we are not aware of a system with similar focus, AppInsight touches upon several active research areas.
Correlating event traces  AppInsight automatically infers causality between asynchronous events in the app execution trace. A number of systems for inferring causality between system events have been proposed, particularly in the context of distributed systems.

LagHunter [50] collects data about user-perceived delays in interactive applications. Unlike AppInsight, LagHunter is focused on synchronous delays such as rendering time. LagHunter requires the developer to supply a list of “landmark” methods, while AppInsight requires no input from the developer. LagHunter also occasionally collects full stack traces, which AppInsight does not do.

Magpie [17] is a system for monitoring and modeling server workload. Magpie coalesces Windows system event logs into transactions using detailed knowledge of application semantics supplied by the developer. On a Windows phone, system-event logs are not accessible to an ordinary app, so AppInsight does not use them. AppInsight also does not require any input from the app developer. Magpie’s goal is to build a model of the system by characterizing the normal behavior. Our goal is to help the developer to detect anomalies.

XTrace [33] and Pinpoint [21] both trace the path of a request through a system using a special identifier attached to each individual request. This identifier is then used to stitch various system events together. AppInsight does not use a special identifier, and AppInsight does not track the request across process/app boundaries. Aguilera et. al. [8] use timing analysis to correlate trace logs collected from a system of “black boxes”. While AppInsight can also use some of these log-analysis techniques, we do not treat the app as a black box, and hence are able to perform a finer grained analysis.

Finding critical path of a transaction  The goal of AppInsight is to detect the critical path in a user transaction. Yang and Miller did early work [84] on finding the critical path in the execution history of parallel and distributed programs. More recently, Barford and Crovella [16] studied critical paths in TCP transactions. While some of our techniques (e.g., building a graph of dependent events) are similar to these earlier works, our focus on mobile apps leads to a very different system design.

Mobile application monitoring  AppInsight is designed to monitor mobile-application performance in the wild. Several commercial products like Flurry [32] and PreEmptive [72] are available to monitor mobile-app usage in the wild. The developer typically includes a library to collect usage information such as number of app launches, session lengths and geographic spread of users. Through developer annotations, these platforms also allow for some simple timing information to be collected. But obtaining detailed timing behavior and critical-path analysis is not feasible with these platforms. To aid with diagnosing crashes, many mobile platforms report crash logs to developers when their application fails. While collecting such data over long term is important [36], it does not necessarily help with performance analysis [5]. Several researchers [73, 68] have studied energy consumption of mobile apps and have collected execution traces for that purpose. Our focus, on the other hand is on performance analysis in the wild.

4.9 Chapter Summary

In this chapter, we presented AppInsight, a system that helps developers to monitor and diagnose their apps in the wild. AppInsight instruments app binaries to collect trace data,
which is analyzed offline to uncover critical paths and exception paths in user transactions. AppInsight is lightweight, it does not require any OS modifications, or any input from the developer. Data from a live deployment of AppInsight shows that mobile apps have a tremendous amount of concurrency, with many asynchronous calls and several parallel threads in a typical user transaction. AppInsight is able to correctly stitch together these asynchronous components into a cohesive transaction graph, and identify the critical path that determines the duration of the transaction. By examining such transactions from multiple users, AppInsight automatically identifies outliers, and sources of variability. AppInsight uncovered several bugs in one of our own app, and provided useful feedback to other developers.
Chapter 5

Adapting To Variable Conditions

In interactive mobile apps, users expect a timely response for their interactions. Responses that arrive within a predictable period of time improve the user experience, whereas the failure to provide consistent response times can have adverse financial implications for even small degradations in response times [43, 18, 77].

AppInsight enables developers to improve response time by identifying bottlenecks and pinpointing the root causes. But, providing consistent response times is still challenging, because there are several variable delays between the start of a user’s request and the completion of the response. These delays include location lookup, sensor data acquisition, radio wake-up, network transmissions, processing in the app, etc. The developer has no control over the variability of some of these delays (e.g. network). To provide consistent performance, the app needs to adapt at runtime to these variable conditions.

In this chapter, we focus on mobile apps that use servers in the cloud for some of their functions, since user interactions that involve communication to a server have long and variable delays. Our goal is to develop a system for app developers to ensure that the end-to-end delay between the initiation of a request and the rendering of the response does not exceed a specified value. The system does not provide hard delay guarantees, but instead makes a best-effort attempt to achieve the delay goal.

Given the desired end-to-end delay, the idea is to allow the server to obtain answers to two questions:

1. **Elapsed time:** How much time has elapsed since the user interaction at the mobile app?

2. **Predicted remaining time:** How much time will it take for the mobile app to receive an intended response over the network and then process it?

The server can use the difference between the desired delay bound and the sum of the elapsed time and predicted remaining time to determine the *work time* for the request. To control the end-to-end delay, the server should adapt and compute its response within the work time.

Although few services are designed with this flexibility today, many are amenable to such adaptation by striking a balance between response quality and work time. For example, search services spawn workers for different content types and aggregate results only from the workers that respond within a deadline [10]; different deadlines lead to different quality of results. Services can also adapt by changing the amount of resources used for request processing, the priority with which response is processed, or the scope of the work (e.g.,
radius for a location-based query). The adaptation mechanisms are service-specific and not the focus of Timecard; Timecard focuses on answering the two questions above.

Answering these questions poses several challenges. Tracking elapsed time requires accurate and lightweight accounting across multiple, overlapping asynchronous activities that constitute the processing of a request on both the mobile device and the server. When the request reaches the server, we must also factor in the clock skew between the client and the server. Inference of this skew is hindered by the high variability in the delay of cellular network links. Estimating remaining time is difficult because it depends on many factors such as device type, network type, network provider, response size, and prior transfers between the client and server (which dictate the TCP window size at the start of the current transfer).

We address these challenges by automatically instrumenting both the mobile app, and the cloud service. To this end, we extend the transaction tracking techniques in Chapter 2 to track the accumulated elapsed time, carrying this value across the stream of thread and function invocations on both the mobile client and server. We also develop a method to accurately infer clock skew, in which probes are sent only when the mobile network link is idle and stable. To predict the remaining time, we train and use a classifier that takes several relevant factors into account, including the intended response size, the round-trip time, the number of bytes already transferred on the connection prior to this response, and the network provider.

We have implemented these ideas in the Timecard system. To study the effectiveness of Timecard, we modified two mobile services to adapt their response quality using the Timecard API. We instrumented the corresponding mobile apps and recruited 20 users to run them on their personal phones for over a month. The results show that Timecard can tightly control the end-to-end response time around a desired user-perceived delay. For instance, in one of the apps, the response time is within 50 ms of the desired delay (1200 ms) 90% of the time.

5.1 Timecard Architecture

Figure 5-1 shows the anatomy of a server-based user transaction in a mobile app. The request starts at time $t_0$. The app does some initial processing, which entails local actions such as reading sensor data and possibly network operations like DNS requests. At time $t_1$ the app makes a request to the server, which reaches the server at time $t_2$. The server processes the request, and sends the response at time $t_3$, which reaches the client at time $t_4$. The app processes the response and renders the final results to the user at time $t_5$. In some cases, transactions have richer patterns that involve multiple calls sequential or parallel to the server. We focus on the single request-response pattern because, as we show in §5.5.1, it is dominant.

The user-perceived delay for this user transaction is the duration $t_5 - t_0$. User-perceived delays for mobile apps vary widely, ranging from a few hundred milliseconds to several seconds (§4.6.4, §5.5.1).

The work time at the server is $t_3 - t_2$. The client’s processing is made up of two parts, $C_1 = t_1 - t_0$ and $C_2 = t_5 - t_4$, which correspond to the duration before the request is sent and the duration after the response is received. We denote the request (“uplink”) and response (“downlink”) network transfer times by $N_1$ and $N_2$, respectively: $N_1 = t_2 - t_1$ and $N_2 = t_4 - t_3$. 

100
Timecard helps app developers control the user-perceived delay for user transactions. It provides an API with two functions for this purpose:

1. `GetElapsedTime()`: Any component on the processing path at the server can obtain the time elapsed since $t_0$.

2. `GetRemainingTime(bytesInResponse)`: At the server, a component can obtain an estimate of $N_2 + C_2$. Timecard provides this estimate as a function of the size of the intended response.

These two functions help control the user-perceived delay. Servers that generate fixed-size responses can infer how much time they have to compute the response by querying for elapsed time and for remaining time with the response size as input. Their work time should be less than the desired user-perceived delay minus the sum of times obtained from those API calls. Servers that can generate variable-sized responses can call this function multiple times to learn how much work time they have for different response sizes, to decide what response they should generate to stay within a given user-perceived delay. The desired user-perceived delay for a transaction is specified by the mobile app developer, based on the responsiveness needs of the app and other factors (e.g., how often the user is refreshing). The API may also be used for other purposes, as discussed in §6.

Determining the elapsed time requires tracking user transactions across multiple asynchronous threads and between the client and server, as well as synchronizing the time between the client and the server. Estimating the remaining time requires a robust way to predict $N_2$ and $C_2$. Figure 5-2 shows the high-level architecture of Timecard, depicting the information flow. Transaction tracking and time synchronization are described in detail in §5.2, while $N_2$ and $C_2$ prediction is covered in §5.3.
5.2 Tracking elapsed time

To track elapsed time, Timecard uniquely identifies each user transaction and tracks information about it, including its start time, in a \textit{transaction context} object (§5.2.1). Timecard also synchronizes the time between the client and the server (§5.2.2). The transaction context is available to any client or server thread working on that transaction. The elapsed time is the difference between the thread’s current time and the transaction’s start time.

5.2.1 Transaction tracking

Transaction tracking is challenging because of the asynchronous programming model used by mobile apps and cloud services. Consider the execution trace of a simple app shown in Figure 5-3. On a user request, the app makes an asynchronous call to obtain its location. After getting the result on a background thread, the app contacts a server to get location-specific data (e.g., list of nearby restaurants). The server receives the request on a listening thread and hands it off to a worker thread. The worker thread sends the response, which is received by the app on a background thread. The background thread processes the response and updates the UI via a dispatcher call, completing the transaction.

To track the elapsed time for this transaction, Timecard passes the transaction’s identity and start time across asynchronous calls, and across the client/server boundary. Timecard instruments the client and the server code to collect the appropriate information, and stores it in a \textit{transaction context} (TC) object (Table 5.1). The instrumentation in Timecard extends the techniques in Chapter 2 in three key aspects: (i) Timecard’s instrumentation tracks transactions on the client, on the server, and across the client-server boundary; (ii) Timecard’s instrumentation enables time synchronization between client and server (§5.2.2); (iii) Timecard collects additional data to enable $N_2$ and $C_2$ prediction.

We now describe how TC is initialized and tracked (§5.2.1), how tracking TC enables Timecard to collect training data for predicting $N_2$ and $C_2$ (§5.2.1), and how TC is reclaimed upon transaction completion.
Request Handler Send
pg
response
Spawn worker
GPS fix Callback
Background thread
Response, Callback
GPS Start
Web Request
Thread
User request
UI Update

Figure 5-3: A location-based app that queries a server.

Table 5.1: Transaction context. The three timestamps are named as per Figure 5-1.

<table>
<thead>
<tr>
<th>Tracked information</th>
<th>Purpose</th>
<th>Set by</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Id</td>
<td>Unique application identifier</td>
<td>Client</td>
<td>Server and Predictor</td>
</tr>
<tr>
<td>Transaction Id</td>
<td>Unique transaction identifier</td>
<td>Client</td>
<td>Client and Server</td>
</tr>
<tr>
<td>Deadline</td>
<td>To calculate remaining time</td>
<td>Client</td>
<td>Server</td>
</tr>
<tr>
<td>$t_3$</td>
<td>To calculate $N_2$ for training data</td>
<td>Client</td>
<td>Predictor</td>
</tr>
<tr>
<td>$t_4$</td>
<td>To calculate $N_2$ and $C_2$ for training data</td>
<td>Client</td>
<td>Client</td>
</tr>
<tr>
<td>$t_5$</td>
<td>To predict $C_2$, and to label training data</td>
<td>Client</td>
<td>Client Predictor</td>
</tr>
<tr>
<td>Entry Point</td>
<td>To predict $N_2$, and to label training data</td>
<td>Client</td>
<td>Server and Predictor</td>
</tr>
<tr>
<td>RTT</td>
<td>To predict $N_2$ and to label training data</td>
<td>Client</td>
<td>Server and Predictor</td>
</tr>
<tr>
<td>Network type</td>
<td>To predict $N_2$ and to label training data</td>
<td>Client</td>
<td>Server and Predictor</td>
</tr>
<tr>
<td>Client type</td>
<td>To predict $N_2$ and to label training data</td>
<td>Client</td>
<td>Server and Predictor</td>
</tr>
<tr>
<td>Size of response from cloud service</td>
<td>To determine when transaction ends</td>
<td>Client</td>
<td>Client</td>
</tr>
<tr>
<td>Pending threads and async calls</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Transaction Context

Timecard identifies all UI event handlers in the app as described in §2.3.2. It instruments the handlers to create a new TC, assigning it a unique ID and timestamp $t_0$. It maintains a reference to the newly created object in the thread’s local storage.

**Tracking a transaction across asynchronous calls:** To pass a reference to the TC from the thread that makes an asynchronous call to the resulting callback thread, Timecard uses the *Detour* objects (§2.3.3). Timecard includes a reference to the TC in the detour object, which allows the thread that executes the callback to access the TC.

**Passing TC from client to server:** When an app makes a request to the server, the client passes some fields in the TC to the server (Table 5.1) by encoding it in a special HTTP header called *x-timecard-request*. To add the header to the HTTP request, Timecard modifies all HTTP request calls in the application.

**Tracking transaction at the server:** Timecard instruments the service entry methods that handle client requests to create a new TC object using the information specified in the *x-timecard-request* header. Timecard then tracks the TC across server threads using the
same techniques as for client threads. **Handling server response and UI updates:** When the response arrives, the client OS invokes a callback method to handle the response. This method has access to the correct TC due to the detouring technique described earlier. The method processes the response and updates the UI via asynchronous calls to a dispatcher.

**Collecting Data to Predict \( C_2 \) and \( N_2 \)**

Transaction tracking also enables Timecard to collect the data to train the \( N_2 \) and \( C_2 \) predictors for subsequent transactions. Figure 5-1 shows that \( N_2 \) and \( C_2 \) may be calculated from \( t_3 \), \( t_4 \), and \( t_5 \). Timecard instruments the server to log \( t_3 \) just before it sends the response to the client. Timecard also records the number of bytes sent in the response. This information, along with transaction id, the device type, client OS, and network provider (Table 5.1) are sent to the predictor.

Timecard instruments the client’s callback handler to log \( t_4 \) as well as the time of the last UI update, \( t_5 \). Once the transaction is complete (§5.2.1), the values of \( t_4 \) and \( t_5 \) along with the transaction id are sent to the predictor. To reduce overhead, this data is sent using a background transfer service on the mobile that schedules the transfer after the app terminates [19].

**Tracking Transaction Completion**

When a transaction completes, Timecard can remove the TC on the client. On the server, Timecard can remove the TC as soon as \( t_3 \) is recorded and sent to the predictor.

A transaction is complete when none of the associated threads are active and no asynchronous calls associated with the transaction are pending. Thus, to track transaction completion on client, Timecard keeps track of active threads and pending asynchronous calls. Because Timecard instruments the start and end of all upcalls, and is able to match asynchronous calls to their callbacks, it can maintain an accurate list of pending threads and asynchronous calls in the TC.

Tracking transaction completion also allows Timecard to detect *idle time* on the client. When there are no active transactions on the client, it means that the app is currently idle (most likely waiting for user interaction). Timecard maintains a list of currently active transactions. When the list is empty, it assumes that the application is “idle”.¹ Timecard uses the application’s idle time in two ways. First, Timecard garbage-collects some of the data structures it needs to maintain to take care of several corner cases of transaction tracking. Second, Timecard uses the start of an idle period to trigger and process time synchronization messages (§5.2.2).

### 5.2.2 Synchronizing time

The timestamps in the TC are meaningful across the client-server boundary only if the client and the server clocks are synchronized. Timecard treats the server’s clock as the reference and implements mechanisms at the mobile client to map its local time to the server’s. The *TimeSync* component code to synchronize the two times is added to the client and server using binary instrumentation. The transaction tracker queries *TimeSync* on the client for a timestamp, instead of the system time.

¹This does not mean that the entire system is idle because other apps may be active in the background.
Before describing our method, we note that two obvious approaches do not work. The first is to run the Network Time Protocol (NTP) [61] on the clients and servers. The problem is that NTP does not handle the significant variability in delay that wireless clients experience; for example, the 3G or LTE interface in idle state takes a few seconds to wake up and transmit data, and in different power states, sending a packet takes different amounts of time. The second approach is to assume that the device can obtain the correct time from a cellular base station or from GPS. Both approaches are problematic: cellular base stations do not provide clients with time accurate to milliseconds, many mobile devices may not have a cellular service, GPS does not work indoors, and also consumes significant energy. For these reasons, Timecard adopts a different solution.

We conducted several measurements to conclude that the clocks on smartphones and servers usually have a linear drift relative to each other, and that the linearity is maintained over long periods of time (§5.5). We assume that the delay between the client and the server is symmetric\(^2\). Given the linearity of the drift and the symmetry assumption, client and server clocks can be synchronized using Paxson’s algorithm [70, 63]. Briefly, the method works as follows:

1. At time \(\tau_0\) (client clock), send an RTT probe. The server responds, telling the client that it received the probe at time \(\tau_1\) (server clock). Suppose this response is received at time \(\tau_2\) (client clock).

2. Assuming symmetric delays, \(\tau_1 = \tau_0 + (\tau_2 - \tau_0)/2 + \epsilon\), where \(\epsilon\) is an error term consisting of a fixed offset, \(c\), and a drift that increases at a constant rate, \(m\).

3. Two or more probes produce information that allows the client to determine \(m\) and \(c\). As probe results arrive, the client runs robust linear regression to estimate \(m\) and \(c\).

However, in case of clients connecting over wireless networks, delays introduced by radio wake-up [46] and by the queuing of on-going network traffic confound this method. These delays are variable, and could be anywhere between a few tens of milliseconds to a few seconds. We develop a new probing technique that is aware of the state of the radio and traffic to produce accurate and robust results. We apply this technique to synchronize the client with each of its servers.

A useful insight is that the ideal time to send RTT probes is soon after a transaction’s response completely arrives from the server, as long as no additional transfers are forthcoming. At this time, the radio will likely be in its high-power (“ready-to-transmit”) state, ensuring that there is no wake-up delay and a lower marginal energy consumption relative to sending a probe when the radio is in any other state. Furthermore, the likelihood of the probe encountering queuing delay at either the client or the base station is also low because mobile devices typically run only one app in the foreground. Background apps are typically not scheduled when a foreground app is active. Base stations maintain per-device queues and implement fair schedulers, so queuing delays are likely to be low at this time. The methods used for client-side transaction tracking know when a transaction has ended and determine when an RTT probe should be sent.

Figure 5-4 shows the performance of our probing method. The graphs are based on data collected from an app that downloads between 1 and 50 Kbytes of data from a server over HSPA and LTE networks. The server and the app were instrumented with Timecard. Apart from

\(^2\)NTP makes this assumption as well. Cellular links can have asymmetric delays, but the difference is typically small. See §5.5 for details.
from the RTT probes sent by Timecard, the app sent its own RTT probes. These additional probes were carefully timed to ensure that they were sent either when the network was busy, or when the network was idle, and the radio was in an idle state (we used the Monsoon hardware power monitor to keep track of the power state of the radio interface). These results show that compared to the probes sent by Timecard, the additional probes experience highly variable round-trip delays, demonstrating the importance of sending probes only when the radio is in a high-power state and when the network is idle.

We conclude the discussion of TimeSync by noting a few additional features of this component. First, Timecard includes an optimization not shown in the graphs above: it collects RTT samples only when the signal strength is above a threshold. The reason is that our data shows that uplink delays are highly variable when the signal strength is low. Second, to minimize the impact on app performance, Timecard computes the linear regression in a background process that runs only when no foreground app is running. Third, the TimeSync component of each app is independent because apps typically use different servers, which may each have a different notion of the current time.

---

Figure 5-4: RTTs of probes from an app to a server with Timecard, when the network is busy, and when the radio is either idle or busy. (Note: There is no high-power idle state in LTE.) Timecard’s probe transmissions strategy results in lower variability.
5.3 Predicting Remaining Time

Timecard’s GetRemainingTime function returns estimates of $N_2$ and $C_2$ for a specified response size. The sum of the two is the total amount of time required to receive and render the response at the client. The estimates are generated by decision tree algorithms that use models built from historical data.

5.3.1 Predicting $N_2$

$N_2$ is the amount of time required to transmit a specified amount of data from the server to the client. $N_2$ depends on a number of factors including the data size, the round-trip time (RTT) of the connection, the number of RTTs required to send the data, the bandwidth of the bottleneck link, and packet loss rate.

Our analysis of traces from over 4000 apps (§5.5.1), shows that (i) 99% of the data transfers are over HTTP (and hence TCP), and (ii) most are quite short – the 90th percentile of the response length is 37 KB, and median is just 3 KB. Hence our focus is to accurately predict duration of short HTTP transfers.

The duration of short TCP transfers over high-bandwidth, high-RTT, low-loss paths is determined primarily by the number of RTTs needed to deliver the data [66]. Modern cellular networks (3G, 4G, LTE) offer exactly such environment: bandwidths can high as 5Mbps, packet losses are rare [82]. However, RTTs can be as high as 200ms [82]. Thus, to predict $N_2$, we need to predict the RTT and estimate the number of RTTs required to transfer a given amount of data.

The number of RTTs required to download a given amount of data depends on the value of the TCP window at the sender when the response is sent. It would seem that the TCP window size and RTT can be easily queried at the server’s networking stack. However, many cellular networks deploy middleboxes [81] that, terminate and split an end-to-end TCP connection into a server-to-middlebox connection and a middlebox-to-client connection. With such middleboxes, the server’s window size or RTT estimate are not useful to predict $N_2$. Other factors that confound the prediction of $N_2$ include the TCP receiver window settings in the client OS, whether TCP SACK is used or not, and other TCP details. Under these circumstances, a method that measures the factors mentioned above and plug them into an analytic TCP throughput formula does not work well. Hence, we use an empirical data-driven model to predict $N_2$. After some experimentation, we settled on a model with the following features:

1. **The response size:** The size of the response, and TCP dynamics (see below), together determine the number of RTTs required.

2. **Recent RTT between the client and server:** We re-use the ping data collected by the TimeSync component (§5.2.2). We also keep track of TCP connection delay for these probes, to account for presence of middleboxes [81].

3. **Number of bytes transmitted on the same connection before the current transfer:** This feature is a proxy for the TCP window size at the sender, which can either be the server or the middlebox, if one is present. We are forced to use this metric because we have no way to measure the TCP window size at a middlebox. However, since TCP sender’s window size generally grows with the number of bytes already sent over the connection, we can use the cumulative number of bytes that were previously transferred on the connection as a proxy for the sender’s TCP window size.
4. **Client OS version and client network provider**: This combined feature is a proxy for the TCP parameters of the client and the middlebox. The client OS version determines the maximum TCP receiver window size and other TCP details. The network provider is the combination of the cellular carrier (Verizon, AT&T, etc.) and network type (LTE, 4G, 3G, etc.). WiFi is a distinct network provider.

Each user transaction provides information about these features and the corresponding response time to the prediction module. The module buckets the observed response-time data into the features mentioned above. Multiple observed response time samples may map to the same bucket, creating a histogram of values for each bucket. The predictor is implemented as a decision tree on these features. It finds the best match among the buckets and returns the median\(^3\) response time value for the bucket. The buckets used by the predictor are updated each time a Timecard-enabled app uploads the feature vector and response time information. Thus, this is an online predictor, with a constantly updating model.

The model used by the \(N_2\) predictor is independent of the application or the service. Thus, we can combine data from multiple Timecard-enabled apps and services to build a more accurate model. We can also bootstrap the model by using offline measurements done by a dedicated measurement app (§5.5).

5.3.2 **Predicting \(C_2\)**

To understand the factors that affect the processing and rendering time on the client after the response is received (i.e. \(C_2\)), we analyzed thirty apps that had 1653 types of transactions. For most transactions, \(C_2\) was highly correlated with the size of the response. Figure 5-5 plots \(C_2\) for a popular transaction in the Facebook application, showing that \(C_2\) is roughly linear in the response length.

\(C_2\) typically includes two components: parsing delay and rendering delay. Many servers send data in the form of JSON, XML or binary (for images). On a mobile device, parsing or de-serializing such data takes a non-trivial amount of time. Our controlled experiments on popular off-the-shelf JSON, XML and image parsers show that, for a given data structure,

\[^3\text{In future, we plan to experiment with other statistics such as the mean or the 90th percentile.}\]
this delay is linear in the data size. We also found that the rendering delay is linear in the data size consumed by the UI which is typically a subset of the response data.

Since the downstream processing is typically computation-bound, \( C_2 \) also depends on the device type and its processing speed. In general, it also depends on whether the current set of apps being run on the device is exhausting memory or CPU resources.

To predict \( C_2 \), we build a decision tree model similar to \( N_2 \) with app id, transaction type, device type, and response data size as the features\(^4\). The \( C_2 \) predictor continuously learns from previously completed transactions. After each transaction, the Timecard client logs the above specified features with a measured value of \( C_2 \) and sends it to the predictor. Thus, like the \( N_2 \) predictor, the \( C_2 \) predictor is also an online predictor. However, unlike the \( N_2 \) predictor, the \( C_2 \) predictor uses numerous models, one per transaction type (which includes the app id), making this predictor difficult to bootstrap. Currently, Timecard requires the app developer to provide rough models for the transaction types in the app, and refines them as more data becomes available. Without developer-provided models, Timecard can simply disable predictions until enough data is available.

5.4 Implementation

Timecard is implemented in C# with 18467 lines of code. It is currently targeted for Windows Phone Apps and .NET services. We do binary instrumentation of both the client- and server-side code. Our instrumentation framework is currently designed for .NET (§2). A majority of the apps in the Windows Phone app store is written in Silverlight. Many web services are powered by .NET (for e.g. ASP.NET) and hosted through IIS. With the popularity of cloud providers such as Amazon Web Services and Azure, developers are able to easily host their services with minimal infrastructure support.

Incorporating Timecard into an app or a service requires little developer effort. We provide Timecard as a Visual Studio package, which can be added into a service or a app project workspace. Once added, it automatically includes a library into the project that exposes the Timecard APIs to the developer. It also modifies the project metadata to include a post-build step where it runs a tool to automatically instrument the built binary. When the instrumented server and the app are deployed, they jointly track transactions, synchronize time, estimate elapsed time, and predict remaining time.

Timecard does not require any modification to Silverlight, the Phone OS, IIS, or the cloud framework.

5.5 Evaluation

In §5.5.1 we demonstrate that network and client delays are highly variable, motivating the potential benefits of Timecard. In §5.5.2 we show that Timecard can successfully control the end-to-end delays for mobile apps. In §5.5.3 we measure the accuracy of the methods to predict \( N_2 \) and \( C_2 \). In §5.5.4 we validate the two key assumptions in the TimeSync component. Finally, we evaluate the overhead of Timecard in §5.5.5.

\(^4\)We currently do not consider memory and CPU utilization.
Is Timecard Useful?

The usefulness of Timecard depends on the answers to three questions. First, how common is the single request-response transaction (Figure 5-1) in mobile apps? This question is important because Timecard is designed specifically for such transactions. Second, how variable are user-perceived delays? Using Timecard, app developers can reduce variability and maintain the end-to-end delay close to a desired value. Third, how variable are the four components \((C_1, C_2, N_1, N_2)\) of the user-perceived delay that Timecard must measure or predict? If these delays are not highly variable, a sophisticated system like Timecard may not be needed.

Common Communication Patterns

We study common communication patterns in mobile apps using the AppInsight and PhoneMonkey datasets (Table 5.2). The AppInsight dataset is based on 30 popular Windows Phone apps instrumented with AppInsight (§4.6.1). We only instrument the clients because we have no control over the servers that these apps use. We persuaded 30 users to use these instrumented apps on their personal phones for over 6 months. Our dataset set contains over 24,000 user transactions that contact a server and 1,653 transaction types. Over 99% of the transactions in the dataset use HTTP-based request-response communication. Moreover, 62% of these transactions involve exactly one request-response communication of the form shown in Figure 5-1.

The dominance of this pattern is further confirmed by our study of 4000 top Windows Phone apps. We instrumented these apps with AppInsight and ran them using the VaranSena UI automation. Across all apps, we obtained over 10,000 unique user transactions with at least one request to a server. We call these traces the PhoneMonkey dataset. Over 80% of PhoneMonkey transactions have the single request-response pattern.

Recall that our network prediction model is geared towards short HTTP transfers (§5.3.1). Figure 5-6 shows the amount of data downloaded in the AppInsight and PhoneMonkey data. The median is only about 3 KBytes, and the 99th percentile is less than 40 KBytes.

To summarize: Timecard addresses the dominant communication pattern in today’s mobile apps.

Variability of User-perceived Delay

Figure 5-7 shows a scatter plot of user-perceived delay and its standard deviation for different types of user transactions in the AppInsight dataset. Each point corresponds to a unique transaction type. We see that the user-perceived delays for a transaction are high—the mean delay is more than 2 seconds for half of the transactions—and also highly variable. This finding highlights the need for a system like Timecard that can control the variability.

---

**Table 5.2: Data sets used for evaluation of Timecard.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Summary</th>
<th>Used in</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppInsight</td>
<td>30 instrumented apps, 30 users, 6 months. Over 24K network transactions.</td>
<td>5.5.1, 5.5.3</td>
</tr>
<tr>
<td>PhoneMonkey</td>
<td>4000 instrumented apps driven by UI automation tool.</td>
<td>5.5.1</td>
</tr>
<tr>
<td>NetMeasure</td>
<td>250K downloads over WiFi/3G/HSPA/LTE over ATT/Sprint/Verizon/TMobile.</td>
<td>5.5.1, 5.5.3</td>
</tr>
<tr>
<td>EndToEnd</td>
<td>2 instrumented apps on 20 user phones sending requests to 2 instrumented services. Over 300K transactions.</td>
<td>5.5.2</td>
</tr>
</tbody>
</table>
Variability of Individual Components

We now show that client-side processing ($C_1$ and $C_2$) and network transfer times ($N_1$ and $N_2$) both contribute to this variability.

**Client processing delays ($C_1$ and $C_2$):** Figures 5-8(a) and 5-8(b) show the absolute values of $C_1$ and $C_2$ and the fraction they contribute to the user-perceived response delay seen in the AppInsight data set. The median delays are around 500 and 300 ms for $C_1$ and $C_2$, while the median fractions are 0.3 and 0.15, respectively. Figure 5-8(c) shows the Coefficient of Variation (CoV) ($\sigma/\mu$) for each unique transaction type. The median values of CoV for $C_1$ and $C_2$ are 0.4 and 0.5, suggesting high variability. We discussed the reasons for the variability of $C_1$ and $C_2$ in §5.2 and §5.3.

**Networking delays ($N_1$ and $N_2$):** The AppInsight data cannot be used to analyze $N_1$ and $N_2$ because it does not have server-side instrumentation. Thus, we built a custom background app for Windows Phone and Android. The app periodically wakes up and repeatedly downloads random amounts of data from a server. Between successive downloads, the app waits for a random amount of time (mean of 10 seconds, distributed uniformly). The download size is drawn from the AppInsight distribution (Figure 5-6). The app and
Figure 5-8: Client processing delays.
the server run the TimeSync method and log $N_1$ and $N_2$.

We ran the app on the personal phones of 20 users, all in the same U.S. city, as well as four Android test devices in two different cities. These phones used a variety of wireless networks and providers such as 3G, 4G (HSPA+), and LTE, on AT&T, T-Mobile, Verizon, and Sprint. The users went about their day normally, providing data across different locations and mobility patterns (indoors and outdoors, static, walking, driving, etc.). The interval between successive wake-ups of the apps was set to anywhere between 1 and 30 minutes depending on user's preference. In all, we collected data from over 250K downloads over a period of one month. We term this the NetMeasure dataset (Table 5.2).

Figure 5-9 shows the CDF of $N_1$ and $N_2$. We see that the delays are both high and highly variable. The median delays are 75 and 175 ms, respectively. 30% of the $N_2$ samples are over 400 ms. Given the values of user-perceived response times in mobile apps (Figure 5-7), these delays represent a substantial fraction of the total.

5.5.2 End-to-End Evaluation

To conduct an end-to-end evaluation of Timecard, we incorporated Timecard into two mobile services and their associated apps. The apps were installed on the primary mobile phones of twenty users, configured to run in the background to collect detailed traces. We term these traces the EndToEnd data set (Table 5.2).

We first describe the services and the associated apps. Next, we show that Timecard helps apps to not exceed a desired end-to-end delay. Finally, we discuss the quality vs. timeliness trade-off for these two services.

The first service is an ad server that delivers contextual ads to apps [64]. The ad server is coupled with a mobile ad control, which is a small DLL that the app developer incorporates into the app. At run time, the mobile ad control scrapes the page displayed by the app for keywords and forwards them to the ad server. The server spawns multiple requests to an ad provider using these keywords. It sorts the received ads according to their relevance and returns the top ad to the app. The returned ad is a small text string, less than 1KB in size. The ad provider needs at least 500 ms to generate one response. By waiting longer, the server can receive additional responses from the ad provider, which can improve the relevance of the returned ad. Hence, there is a trade-off between server work time and the ad quality. The ad server uses the API described in §5.1 to determine how long the service

\begin{figure}
\centering
\includegraphics[width=\textwidth]{network_delays.png}
\caption{Network transfer delays.}
\end{figure}
Figure 5-10: User-perceived delays for two apps. With Timecard, delays are tightly controlled around the desired value.

Figure 5-11: Elapsed time \((C_1 + N_1)\) from two apps.
should wait before sending an ad to the client. Note that the specified delay is not a hard deadline; Timecard tries to keep the actual delay around the specified value, seeking to reduce the delay variance around that value.

We built a simple app and added the ad control to it. The app wakes up at random times and feeds randomly selected keywords based on data in [64] to the ad control. We set the desired end-to-end delay for fetching ads to be 1.2 seconds.

The second service is a Twitter analysis service, with an associated mobile app that has been in the Windows Phone store for over 2 years. The app lets the user specify a keyword, which it sends to the analysis service. The service fetches recent tweets for the keyword, categorizes them into positive and negative tweets (sentiment), and finally sends an aggregated sentiment score to the app.

We modified the app to specify a desired delay of 1.1 seconds in addition to specifying the keyword. The server uses Timecard to decide how many tweets to fetch and analyze, given the desired delay. The quality of the response (sentiment analysis and the aggregated score) improves with the number of tweets, but fetching and analyzing more tweets takes more time. If more tweets are fetched, the size of the response sent to the app increases as well. The service sends between 8 KB to 40 KB of data per request. The app simply parses and renders the response.

Figure 5-12: Deadline control errors: discrete work time in one app and $N_2$ prediction errors.
Because of restrictions imposed by Twitter's web API, the service can only fetch and process tweets in multiples of 100, so the work time can be adjusted only in steps of roughly 150 ms. As a result, we cannot always meet the deadline precisely, but the server attempts to ensure that the user-perceived delay is smaller than the 1.1-second deadline. We precomputed the expected work times for fetching and analyzing different numbers of tweets by separately profiling the service.

We bootstrapped the $N_2$ predictor for both services using the NetMeasure data set. We bootstrapped the $C_2$ predictor using offline measurements.

Figure 5-10 shows that with Timecard these two apps achieve user-perceived delays that are tightly distributed around the desired value. This result is significant because the upstream elapsed time when the request hits the server is highly variable, as shown in Figure 5-11. Over 90% of the transactions are completed within 50 ms of the specified deadline for ad control.

The difference between the observed and the desired delay is due to two main factors. For the Twitter analysis service, the work time is limited to be a multiple of 150 ms. Figure 5-12(a) shows that this causes 80% of the transactions to finish before the deadline, and over half the transactions to finish 50 ms early. The error in $N_2$ and $C_2$ prediction is the other main reason for the observed delay being different than the desired delay. Figure 5-12(b) shows that the median error in $N_2 + C_2$ is only 15 ms for the ad control app, because the service returns a small amount of data for each request. The median error is higher (42.5 ms) for the Twitter analysis app. TimeSync error also likely contributes to the downstream prediction error; unfortunately we have no way of measuring its precise impact.

As the two services described above try to meet the end to end deadline, they trade-off quality of results for timeliness of response.

Figure 5-13(a) shows the trade-off between the ad server work time and probability of fetching the best ad. Recall that we had set the total deadline to be 1.2 seconds. Thus, the best ad is the ad that would have been top rated if the server had spent the entire 1.2 seconds on the job. If the server spends less time, it may not always find the best ad. Using trace data from 353 apps, 5000 ad keywords [64], and about 1 million queries to the ad server, we calculate the probability that the best ad is found for various work times. As one might expect, the probability increases as the server spends more time. Similarly, Figure 5-13(b) shows the trade-off between fetching and analyzing different number of tweets (mapped to average server work time) and the quality of sentiment analysis. The data is based on over 150,000 tweets for 100 popular keywords. We see that as the server spends more time to fetch and analyze more tweets, the error in aggregated sentiment score compared to fetching and analyzing the maximum of tweets (1500) from twitter is reduced.

We stress that these trade-off curves are specific to each service; we do not claim that the above curves are representative in any manner.

### 5.5.3 Prediction Accuracy

**Accuracy of $N_2$ Prediction:** We evaluate accuracy of $N_2$ prediction using the NetMeasure dataset (Table 5.2). We randomly split the data into two halves for training and testing. Figure 5-14(a) shows the CDF of absolute errors in the prediction. The median error is 23 ms; the 90th percentile error is 139 ms. To dig deeper, we look at WiFi and cellular links separately. We find that our prediction is more accurate for WiFi (median error 11.5 ms median, 90th percentile 31 ms) than it is for cellular networks (median 31 ms, 90th percentile 179 ms). Some of the longer tail errors (>100 ms) for cellular networks are due to
Figure 5-13: Trade-off between server work time and quality of results.
Figure 5-14: Accuracy of \( N_2 \) and \( C_2 \) prediction
radio wake-up delays on the downlink. In certain device models and carriers, the radio does not go to highest power state during upload, since upload transfers (i.e. client requests) are assumed to be small. Full wake-up happens only when the download begins.

The data size also has an impact on prediction delay, due to complex interactions between server TCP state, middlebox TCP state, and client TCP parameters. For smaller data sizes, these interactions do not matter as much, so the prediction error is low when we download less than 37 KBytes (median 17 ms, 90th percentile 86 ms). Recall that in the AppInsight data set, 37 KBytes represents the 90th percentile download size (Figure 5-6).

Recall from §5.3.1 that we use the amount of data already transferred on the connection as a coarse way of modeling the TCP window behavior at the middlebox or the server. Figure 5-14(b) shows that it is important to include this feature in the model. Without the cumulative data sent, the median error in $N_2$ prediction is 54% higher, and almost double for the 90th percentile.

Accuracy of $C_2$ prediction: We use the AppInsight data set to evaluate the accuracy of $C_2$ predictor. In 30 apps, we identity 100 transaction types that have at least 20 transactions each from different users or different sessions. We use half the data for training and the other half for testing. Figure 5-14(c) plots the absolute error in $C_2$ prediction. The median error is 8 ms, but the 90th percentile error is 261 ms. When normalized for transaction duration, the median error is 4.6%, while 90th percentile is 22%. Both the percentage and absolute errors are low for shorter transactions. The graph shows that for transactions with $C_2 < 1$ second, the 90th percentile $C_2$ prediction error is 100 ms (10%). It also shows that $C_2$ predictor must take the size of the downloaded data into account. Without it, the median error is over 150 ms.

5.5.4 TimeSync

Our TimeSync method assumes that the clock drift is linear and that the uplink and downlink delays are symmetric. We now test these hypotheses.

We connected a smartphone to a desktop machine and sent TimeSync RTT probes from the smartphone to the desktop over the low delay USB link. We found that the combined drift between the smartphone clock and desktop clock is linear, and stayed linear over several days. We repeated this experiment on many different smartphone models and obtained
similar results. Figure 5-15 shows the clock drift on seven different smartphones over a day. A simple linear regression fits the data and the mean error is 0.8 ms.

Cellular networks can have asymmetric uplink and downlink delays [46]. To estimate the asymmetry, we connected a smartphone to a desktop and sent probes from the desktop through the phone's cellular connection (tethering), back to the desktop's Ethernet connection. By using a single clock to measure uplink and downlink delays, we can measure the difference between the two (i.e., the asymmetry). We find that for three LTE networks, the difference between the uplink and downlink delay is less than 5 ms. But on 3G networks, the difference can be as high as 30 ms. The error in time synchronization can be as high as this difference, which impacts the accuracy of the elapsed time estimation. Thus, highly asymmetric links can cause Timecard to miss the overall deadline. We also find that low signal strength greatly impacts the cellular uplink, making the probe delays asymmetric. Thus, we do not collect probes samples when the signal strength is low.

5.5.5 Overhead

To quantify the overhead of Timecard, we use an HTC Mazaa running Windows Phone 7.1 as client and an HP Z400 2.8 GHz dual-core with 16 GB RAM as server.

**App run time:** The impact of Timecard on app's run time is negligible. The average overhead of tracking an edge in the transaction graph is 50 μs. For the apps in the AppInsight data set, we estimate that the average total increase in app's run time would be 2 ms, which is less than 0.1% of the average transaction length. Overhead of sending and processing of RTT probes is minimal, due to various optimizations described in (§5.2.2). Timecard increases app launch time slightly (2 ms), since it needs to initialize various data structures. Regular garbage collection and bookkeeping of various data structures is done during app idle time. The AppInsight data set shows that all apps have more than 10% idle time, which is sufficient for our needs.

**Service run time:** The overhead of Timecard at the server is small. The average time required for tracking an edge is less 10 μs. Overall, for the two services we instrumented, Timecard adds less than 0.1 ms to processing of each request.

**Memory:** Timecard consumes between 20 KB to 200 KB of additional memory to keep track of various data structures. Since the average memory consumption of apps in the AppInsight data set is 50 MB, the memory overhead of Timecard is less than 1%. On the server, the memory overhead of Timecard is negligible.

**Network:** Timecard consumes network bandwidth during app execution to send transaction context to server (§5.2.1) and to send RTT probes for TimeSync (§5.2.2). It also sends log data to the predictor to improve the prediction models. The size of the extra header is only 50–100 bytes. In rare cases, however, adding extra bytes can increase the request size just enough so that TCP incurs an extra round trip to send the request. TimeSync probes are small packets and transfer only a few bytes of data. The amount of data sent to predictor per transaction is just 20 bytes. Furthermore the training data is uploaded using background transfer. The total network overhead is less than 1% for the apps we instrumented.
The server incurs roughly the same network overhead. Most cloud services are deployed in well-provisioned data centers, and the marginal overhead is insignificant.

**Battery:** The battery overhead of Timecard that results from additional network usage is worth discussing; the CPU overhead is small. We time our RTT probes to avoid a radio wake-up power surge (§5.2.2). The battery impact of the few additional bytes sent in each request header is small. Thus, although we have not actually measured the marginal battery consumption, we see no reason why it would be significant.

### 5.6 Discussion and Limitations

**Limitations of the $N_2$ predictor:** Our approach for predicting $N_2$ has several limitations. First, for large transfer sizes, the number of RTTs matters less than the bottleneck rate of the connection. This limitation does not matter much for our purposes, because our focus is on request-response interactions (the common case for mobile apps). Second, a cellular provider could arbitrarily alter middlebox parameters, so the learning has to be continuous and may require retraining. In our experiments we observed consistent middlebox behavior for over a month, but that behavior may not always hold. Third, our model does not use the client's location as a feature. A network provider could deploy differently-behaving middleboxes in different areas, reducing the predictor's effectiveness. If that is observed, we would need to include the location as a feature. Fourth, our predictor depends on recent RTT samples. For geo-replicated servers, we could end up measuring RTT to a different server than the one client eventually downloads data from. If that happens, our prediction can be erroneous. In practice, we believe that this situation is uncommon because of the nature of replica selection algorithms.

**Complex transactions:** We focused on user transactions that included a single request to a cloud service. However, Timecard can be extended to more complex patterns (parallel or sequential requests to servers, complex dependencies between server requests, etc.) as well. For the GetElapsedTime() call, Timecard needs to ensure that the right timestamp is used with the right server. Extending GetRemainingTime() is more complex, and may require the developer to apportion budgets among multiple servers.

**Privacy and security:** Timecard does not collect any information that app developers cannot collect for themselves today. However, any logging and tracing system must carefully consider privacy implications, a topic for future work. For example, it is worth investigating the smallest amount of information that Timecard needs to log to function effectively. There are other security implications as well. For example, clients may manipulate the transaction data sent to the server, so that they get the “best possible” service.

**Server processing time:** We have not shown that the server processing time ($S$ in Figure 5-1) is a significant portion of the user-perceived delay for popular mobile apps. To do so, we would need to instrument several third-party mobile services\(^5\), which is a challenging, if not impossible, task. We also note that while several services such as search offer a clear trade-off between processing time and quality of results, such a trade-off is not possible for all services. However, even such services can use Timecard, as we discuss in §6.

\(^5\)Without such instrumentation, we cannot tease apart $S$ and $N_2$. 

121
5.7 Related Work

Mobile app monitoring and analysis: AppInsight is an analytic tool, focused on app performance. In contrast, Timecard allows developers to manage user-perceived delays. SIF [44] is another system closely related to AppInsight. Unlike AppInsight, SIF includes a programming framework to help the developer instrument selected code points and paths in the code. Like AppInsight, SIF focuses on client performance only. Other systems for monitoring mobile apps have primarily focused on profiling battery consumption [73, 68]. Flurry [32] and PreEmptive [72] provide mobile app usage monitoring. These systems do not provide tools for managing end-user response times, nor handle server-based mobile apps.

Predicting mobile network performance: A number of recent studies have focused on mobile network performance; we discuss two recent ones that have focused on prediction. Sprout [82] is a UDP-based end-to-end protocol for mobile applications such as videoconferencing that require both low delays for interactivity and high throughput. Sprout uses a model based on packet inter-arrival times to predict network performance over short time periods. Proteus [83] passively collects packet sequencing and timing information using a modified socket API, and uses a regression tree to predict network performance over short time periods. Proteus is also primarily applicable to UDP flows. Timecard can use any improvements in techniques to predict mobile network performance. Our implementation does not borrow from either Sprout or Proteus, however, because our primary focus is on apps that use TCP.

Server performance monitoring: The literature on monitoring transactions in distributed systems goes back several decades. We highlight three recent proposals. Magpie [17] monitors and models server workload, but unlike Timecard, has no client component. XTrace [33] and Pinpoint [21] trace the path of a request using a special identifier. Timecard uses similar techniques, with a focus on managing end-to-end delays.

Data center networking: Much effort has been devoted to understanding and minimizing delays in data centers that host delay-sensitive services. This work includes new network architectures [40, 9, 51], new transport protocols [10, 78], and techniques to rearrange computation and storage [4, 65]. None of these proposals is concerned with managing end-to-end deadlines.

Time synchronization: A number of innovative proposals for time synchronization in various domains, such as the Internet [70, 63, 61], wireless sensor networks [27, 53], and large globally-distributed databases [23] have been developed. Timecard currently uses algorithms proposed in [70, 63], but can benefit from any appropriate advances in this area.

5.8 Chapter Summary

Timecard helps manage end-to-end delays for server-based mobile apps. For any user transaction, Timecard tracks the elapsed time and predicts the remaining time, allowing the server to adapt its work time to control the end-to-end delay of the transaction. Timecard incorporates techniques to track delays across multiple asynchronous activities, handle time skew between client and server, and estimate network transfer times. Our experiments
showed that Timecard can effectively manage user-perceived delays in interactive mobile applications. These results suggest that Timecard’s API functions may be used by developers to re-design their apps and services to achieve good trade-offs between the quality of responses to requests and user-perceived delay.
Chapter 6

Conclusion

We conclude the dissertation with a summary of our contributions and a discussion of future work.

6.1 Summary

This dissertation presented three closely related systems to help mobile app developers improve the performance and reliability of their apps:

- **VanarSena**, a testing system that enables developers to uncover app failures before the app is deployed in the wild.
- **AppInsight**, a monitoring system that enables developers to understand app performance and failures in the wild.
- **Timecard**, a system that enables developers to adapt their apps at runtime to provide consistent performance in the face of varying conditions in the wild.

These systems are built on top of a binary instrumentation framework that can automatically rewrite any binaries. The systems require minimal developer effort to use and requires no modification to the OS or runtime.

This dissertation introduced the notion of *user transactions* in mobile apps. A user transaction represents the execution of a single user activity in the app. It begins with a user interaction of the UI, and ends with completion of all synchronous and asynchronous tasks in the app that were triggered by the interaction. Tracking the execution of the app as a set of user transactions enables us to better understand and reason about user-perceived performance and failures.

We described techniques to efficiently test a mobile app. We introduced the principle of "greybox" testing for mobile apps by instrumenting the app and getting detailed insights into the app runtime behavior. This helped us significantly reduce testing time.

We presented techniques to effectively manage user-perceived delays in mobile apps. We introduced the idea of *critical paths* in user transactions and provided feedback to the developer about optimizations needed for improving user-perceived delays. We demonstrated new techniques to reduce performance variability and tightly control end-to-end user-perceived delays around a specified value for server-based mobile apps.

The systems presented in this dissertation provided detailed insights into real-world crashes and performance in mobile apps. We studied 25 million crash reports from more
than 100,000 apps to design our testing framework. We uncovered thousands of bugs in thousands of apps deployed in the app store. The real-world deployments provided detailed understanding of performance bottlenecks, components of delay, and performance variability in apps.

### 6.2 Future Work

The dissertation opens up many avenues for future research in testing, monitoring, and adapting mobile applications.

**Replay Testing:** VanarSena cannot adequately test apps and games that require complex free-form gestures or specific order of inputs. For these apps, trace replay may be a more appropriate testing strategy than randomized monkey actions. Collection and replay of traces in mobile apps has several challenges. To do deterministic replay, in addition to recording user actions, we need to record uncontrolled inputs such as data from network and random numbers. We also need efficient techniques to replay them with overhead.

**Drill-Down Monitoring:** AppInsight collects just enough data from the wild to reconstruct user transactions. It identifies critical paths at the granularity of asynchronous edges. When the app has long running threads, the data collected is not enough to pinpoint the synchronous bottlenecks. Instrumenting and collecting synchronous API delays from the wild can incur prohibitive overhead. To solve this problem, we could build a drill-down monitoring system. After identifying the critical path, the system will iteratively drill-down on that particular path to collect more information. This will help provide insightful feedback to developers. Such drill-down techniques can also be used to collected more information about crashes in the wild.

There are several challenges in building such a monitoring framework. Since the application cannot be updated on-the-fly, the app needs to be instrumented upfront for drill-down monitoring. To do this, new instrumentation techniques are needed to keep the overhead low. Also, data needs to be efficiently collected across all users and the collection pipeline needs to be continuously adapted based on the resources available.

**Bridging Monitoring and Testing:** Currently, VanarSena operates independent of AppInsight. We could use data collected from AppInsight to drive the testing in VanarSena. This will help VanarSena focus on testing features that are heavily used in the wild and emulate conditions from the wild more accurately. The test inputs can also be learnt across apps.

**Adapting Servers:** The APIs in Timecard can be used even by services that cannot control the response quality versus processing time trade-off. For instance, a server can use the APIs to prioritize the order in which requests are served, so that requests most in danger of missing user-perceived delay deadlines are served first. The server can also allocate different amount of resources to requests based on their deadline. A component on the mobile device may use the elapsed time to decide not to contact the server but use a cached response if the elapsed time is already too long. Alternatively, if the request involves the delivery of speech or sensor samples, it can adjust the sampling rate depending on the elapsed time.
Finding the Right Protocol Parameters: Mobile apps differ widely in their runtime behavior and protocol parameters need to adapt to application behavior to perform efficiently. For instance, our recent analysis on thousands of mobile apps shows that mobile apps have very different network access characteristics. And there is no single network protocol setting that can perform well for all apps. There is need for protocols to adapt based on the behavior of the application to maximize performance. We believe that, it applies to other system components such as caching, garbage collection, radio wakeup etc. To this end, we want to explore the idea of auto-tuning apps where we can automatically analyze the runtime behavior of an app and rewrite the protocol parameters based on application behavior. We could use a framework similar to the VanarSena that can dynamically analyze the behavior of the app based on conditions in the wild and find the optimal parameter settings for the app.
Bibliography


[76] Luigi Rizzo. Dummynet.


