AppInsight: Mobile App Performance Monitoring in the Wild

Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan,
Ian Obermiller, Shahin Shayandeh
Microsoft Research

Abstract— The mobile-app marketplace is highly competitive. To maintain and improve the quality of their apps, developers need data about how their app is performing in the wild. The asynchronous, multi-threaded nature of mobile apps makes tracing difficult. The difficulties are compounded by the resource limitations inherent in the mobile platform. To address this challenge, we develop AppInsight, a system that instruments mobile-app binaries to automatically identify the critical path in user transactions, across asynchronous-call boundaries. AppInsight is lightweight, it does not require any input from the developer, and it does not require any changes to the OS. We used AppInsight to instrument 30 marketplace apps, and carried out a field trial with 30 users for over 4 months. We report on the characteristics of the critical paths that AppInsight found in this data. We also give real-world examples of how AppInsight helped developers improve the quality of their app.

1 INTRODUCTION

There are over a million mobile apps in various app marketplaces. Users rely on these apps for a variety of tasks, from posting mildly amusing comments on Facebook to online banking.

To improve the quality of their apps, developers must understand how the apps perform in the wild. Lab testing is important, but is seldom sufficient. Mobile apps are highly interactive, and a full range of user interactions are difficult to simulate in a lab. Also, mobile apps experience a wide variety of environmental conditions in the wild. Network connectivity (Wi-Fi or 3G), GPS-signal quality, and phone hardware all vary widely. Some platform APIs even change their behavior depending on the battery level. These diverse conditions are difficult to reproduce in a lab. Thus, collection of diagnostic and performance trace data from the field is essential.

Today, there is little platform support for tracing app performance in the field. 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 [1], and this data does not help diagnose performance problems. Analytics frameworks such as Flurry [8], and Preemptive [16] 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 interactive. To keep the UI responsive, developers must use an asynchronous programming model with multiple threads to handle I/O and processing. 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 paper, 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 instruments mobile apps mainly by interposing on event handlers. The performance data collected in the field is uploaded to a central server for offline analysis. The design of AppInsight was guided by three principles. (i) **Low overhead:** We carefully select which code points to instrument to minimize overhead. (ii) **Zero-effort:** We do not require app developers to write additional code, or add code annotations. Instrumentation is done by automatically rewriting app binaries. (iii) **Immediately deployable:** We do not require changes to mobile OS or runtime.

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

```
void btnFetch_Click(
    object obj, 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: Example of asynchronous coding pattern

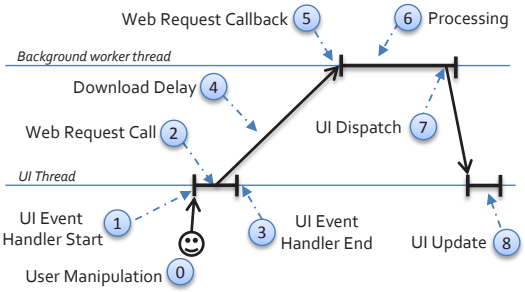

Figure 2: Execution trace for the code in Figure 1.

variation in the environment. In § 8.2, we will discuss how this feedback helped developers improve the quality of their app.

This paper makes two main contributions. First, we describe several innovative techniques that automatically instrument mobile apps to monitor user transactions with minimal overhead. These techniques are embodied in the current implementation of AppInsight. Second, we present results from a real-world study of 30 Windows Phone apps that we instrumented using AppInsight.

2 Mobile App Monitoring

We now discuss the typical asynchronous programming pattern used in mobile apps, and the challenge it presents for monitoring performance and failures.

Mobile apps are UI-centric in nature. In modern UI programming frameworks [6, 15], 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 [15], 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, which shows the execution trace for a simple code snippet in Figure 1. In the figure, horizontal line segments indicate time spent in thread execution, while arrows between line segments indicate causal relationships between threads.

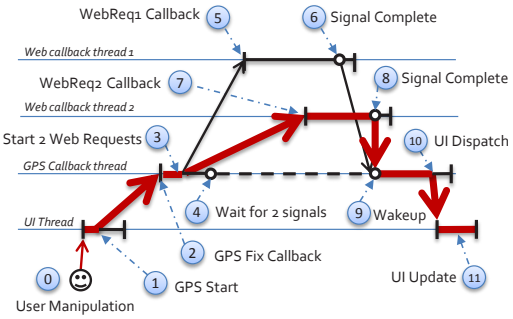

Figure 3: Execution trace of a location-based app.

(0) The user starts the transaction by clicking a button; (1) the OS invokes the event handler (`btn-Fetch_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.

Real apps, of course, are much more complex. For example, (*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 3 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 3, the bottleneck path involves the second web request, which took longer to complete.

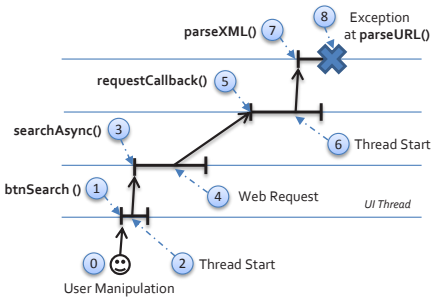

Figure 4: Execution trace of an app crash.

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 4. 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. This is one of the goals of AppInsight, as we discuss next.

3 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.

User transaction: A user transaction begins with a user manipulation of the UI, and ends with completion of all synchronous and asynchronous tasks (threads) in the app that were triggered by the manipulation. For example, in Figure 2, the user transaction starts when the user manipulation occurs and ends when the `updateUI` method completes. A user transaction need not always end with a UI update. For example, a background task may continue processing past the UI update, without impacting user-perceived latency. The notion of user-perceived latency is captured in our definition of *critical path*, which we turn to next.

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. Informally, the critical path starts with a user manipulation event, and ends with a UI update event. In Figure 2, 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 3, 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.

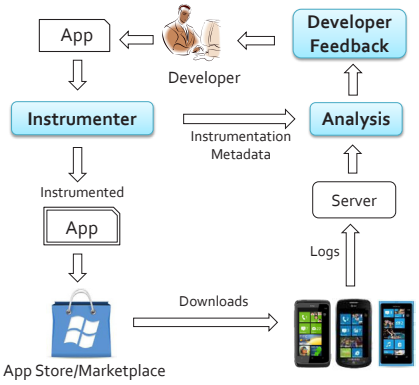

Figure 5: AppInsight System Overview

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 § 6.2.

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 4, (0) to (8) is the exception path. The exception path points the developer to the user manipulation 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 AppInsight Design Overview

Figure 5 shows the architecture of AppInsight. The app binary is instrumented using an instrumentation tool (the instrumenter) that we provide. The developer only needs to provide the instrumenter with app binaries; no other input or source code annotation is needed.

The instrumenter leverages the fact that phone apps are often written using higher-level frameworks and compiled to an intermediate language (byte code). Our current implementation is designed for apps written using the Silverlight framework [15], compiled to MSIL [13] byte code. MSIL preserves the structure of the program, including types, methods and inheritance information.

Silverlight is used by a vast majority of the apps in the WP7 marketplace. AppInsight requires no special support from the Silverlight framework.

When users run the instrumented app, trace data is collected and uploaded to a server. We use the background transfer service (BTS) [18] 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 (§ 7).

5 INSTRUMENTATION

We now describe our instrumenter in detail. Its goal is to capture, with minimal overhead, the information necessary to build execution traces of user transactions and identify their critical paths and exception paths.

A number of factors affect the performance of mobile applications: user input, environmental conditions, etc. Even 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 give useful feedback to the developer.

Figures 3 and 4 indicate that, we need to capture six categories of data: (*i*) when the user manipulates 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 was updated; (*vi*) any unhandled exceptions. Apart from this, we also capture some additional data, as discussed in § 5.7.

To collect the data, we instrument the app in three steps. First, we read the app binary and assign a unique identifier to all methods in the app code and to system calls. Each call site is considered unique; if X is called twice, each call site gets its own identifier. This mapping is stored in a metadata file and uploaded to the analysis server for later use.

Second, we link two libraries to the app – a Detour library and a Logger library (see Figure 6). The Detour library is dynamically generated during instrumentation. It exports a series of detouring functions [11], which help attribute callback executions to the asynchronous calls that triggered them. The Logger library exports several logging functions and event handlers that insert trace records into a memory buffer. Each record is tagged with a timestamp and the id of the thread that called the logging function. The buffer is flushed to stable storage to prevent overflow as needed. When the app exits, the buffer is scheduled for upload using BTS.

Finally, we instrument the app binary with calls to methods in the Logger 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 1, and the corresponding transaction diagram in Figure 2 as a running example.

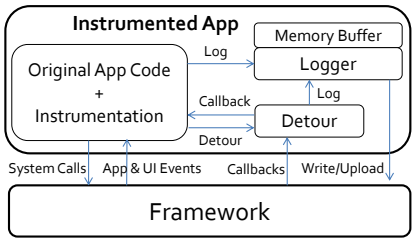

Figure 6: Structure of Instrumented App

5.1 Capturing UI Manipulation events

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 is always a ManipulationStarted event, and the last is always 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 1, `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 logger library exports handlers for ManipulationStarted and ManipulationEnded events, which we add to the app's code. The handlers log the times of the events, which allows us to match the UI manipulation to the right app handler for that UI input.

5.2 Capturing thread execution

The horizontal line segments in Figure 2 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 and uploading 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, 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. For our purpose, it is enough to log 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.

```
void btnFetch_Click(
    object obj, RoutedEventArgs e) {
  + Logger.LogUpcallStart(5);
  var req = WebRequest.Create(url);
  * Detour dt = DetourFactory.GetDetour(reqCallback, 7);
  * Logger.LogCallStart(7);
  req.BeginGetResponse(dt.Cb1, null);
  * Logger.LogCallEnd(7);
  + Logger.LogUpcallEnd(5);
}
void reqCallback(IAsyncResult result) {
  + Logger.LogUpcallStart(19);
  /* Process */
  * Detour dt = DetourFactory.GetDetour(updateUI, 13);
  * Logger.LogCallStart(13);
  UIDispatcher.BeginInvoke(dt.Cb2);
  * Logger.LogCallEnd(13);
  + AppInsight.LogUpcallEnd(19);
}
void updateUI() {
  + Logger.LogUpcallStart(21);
  /* Update UI */
  + Logger.LogUpcallEnd(21);
}
```

Figure 7: Instrumented version of the code in Figure 1. The actual instrumentation is done on MSIL byte code. We show decompiled C# code for convenience.

We identify all potential upcall methods using a simple heuristic. 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 1, a reference to `reqCallback` is passed to the `BeginGetResponse` system call. The MSIL code for creating a delegate has a fixed format [13], 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[1].

We capture the start and end times of all potential upcalls, along with the ids assigned to them, as shown in Figure 7. The instrumentation added for tracking potential upcalls is prepended by '+'. All three methods in the example are potential upcalls and thus instrumented[2].

While this technique is guaranteed to capture all upcalls, it may instrument more methods than necessary, imposing unnecessary overhead. This overhead is negligible, compared to the savings achieved (§ 8.3).

5.3 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, 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 was made, e.g., in Figure 1, the `BeginGetResponse` call is an asynchronous system call. Second, we need to log when the callback started executing as an upcall. We have already described how we track the start of upcall execution. Third, we need to connect the beginning of callback execution to the right asynchronous call.

```
public class DetourFactory {
  ...
  public static Detour GetDetour(
          Delegate d, int callId)  {
    int matchId = getUniqueId();
    Logger.LogAsyncStart(callId, matchId);
    return new Detour(d, matchId);
  }
}
public class Detour {
  int matchId; Delegate originalCb;
  public Detour(Delegate d, int matchId)  {
    this.originalCb = d; this.matchId = matchId;
  }
  public void Cb1(IAsyncResult result) {
    Logger.LogCallbackStart(this.matchId);
    Invoke(this.originalCb);
  }
  public void Cb2() {
    ...
  }
}
```

Figure 8: Detour library

We solve the first problem by assuming that any system call that accepts a delegate as an argument, is an asynchronous call. This simple heuristic needs some refinements in practice, which we will discuss in § 5.3.1.

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 7 shows instrumented code for the example in Figure 1. Instrumentation used for detour is tagged with '*'. Figure 8 shows relevant code inside the Detour library. We add instrumentation as follows.

(*i*) We identify the system call `BeginGetResponse` as an asynchronous call. The instrumenter has assigned a call id of 7 to this call site. We log the call site id, and the start and end time of the call[3].

(*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 8).

(*iii*) We instrument the call site to call `GetDetour` to generate a new instance of the `Detour` object. This ob-

[1] Certain UI handlers are passed to the system differently. We identify them as well – we omit details due to lack of space.

[2] The method btn_FetchClick is a UI handler, and a pointer to it is passed to the system elsewhere.

[3] Async calls typically return almost immediately. We log both start and end of these calls not to collect timing data, but because such bracketing makes certain bookkeeping tasks easier.

```
Thread t = new Thread(foo);
...
...
t.Start();
```

Figure 9: Delayed callback

ject stores the original callback, and is assigned a unique id (called *matchId*) at runtime. This `matchId` 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 8 that the `GetDetour` method logs the beginning of an asynchronous call using the `LogAsyncStart` function of the Logger library. Similarly, the beginning of the callback is logged by the `LogCallbackStart`, which is called from `cb1`, just before the original callback is invoked. These records, and the `UpcallStart` record of the original callback method are linked by the `matchId`, the call site id, and their thread ids, allowing us to attribute the callback to the right asynchronous call. We show an example in § 5.8.

Figure 7 also shows another example of detouring. The `UpdateUI` method is a callback for the `BeginInvoke` method of the `UIDispatcher`, and hence is detoured.

5.3.1 Refining async-call identification heuristic

The simple heuristic used to determine which system calls are asynchronous calls, needs two refinements in practice. First, some system calls may invoke the supplied callback synchronously. This can be easily detected using thread ids in the trace. The second problem is more complex. Consider Figure 9. 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. The simple heuristic would incorrectly match the callback to the call site of the constructor, instead of `Thread.Start`. We use domain knowledge about Silverlight system libraries to solve the problem. We know that the callback function is always invoked from `Thread.Start`. We log the id of the thread object at the constructor, and also at `Thread.Start`. The object ids, and the detour log described above allow us to match the callback to the `Thread.Start` call. We handle event subscriptions in a similar manner.

5.4 Capturing Thread Synchronization

Silverlight provides a set of methods for thread synchronization. The thread waits on a semaphore (e.g., Monitor.Wait(obj)), and is woken up by signaling that semaphore (e.g., Monitor.Pulse(obj)). We log calls to these functions and the identities of semaphore objects they use. These object ids can be used to determine the causal relationship between synchronization calls. Waiting on multiple objects, and thread join calls are handled similarly. Threads can also synchronize using shared variables. We will address this in § 9.

RecordId	Records	ThreadId
1	UIManipulationStarted	0
2	MethodStart(5)	0
3	CallStart(7)	0
4	AsyncStart(7, 1)	0
5	CallEnd(7)	0
6	MethodEnd(5)	0
7	UIManipulationEnded	0
8	CallbackStart(1)	1
9	MethodStart(19)	1
10	CallStart(13)	1
11	AsyncStart(13, 2)	1
12	CallEnd(13)	1
13	MethodEnd(19)	1
14	CallbackStart(2)	0
15	MethodStart(21)	0
16	MethodEnd(21)	0
17	LayoutUpdated	0

Table 1: Trace of code in Fig. 7. The UI thread id is 0.

5.5 Capturing UI updates

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 *single* LayoutUpdated event is raised when the upcall ends. The Logger library exports a handler for this event, which we add to the app code. The handler logs the time this event was raised.

5.6 Capturing unhandled exceptions

When an unhandled exception occurs in the app code, the system terminates the app. Before terminating, the system delivers a special 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 logger library exports a handler for this event, which we add to the app code.

5.7 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 completness 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.

5.8 Example trace

Table 1 shows the trace generated by the instrumented code in Figure 7. Records 1 and 7 show a UI Manipulation event. They encompass an upcall (records 2-6) to the method `btnFetch_Click`. As described in § 5.1, we attribute this upcall to UI manipulation.

This method makes the asynchronous system call `BeginGetResponse` (record 4), the callback of which is

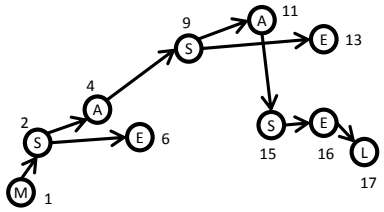

Figure 10: Transaction Graph for the trace in Table 1.

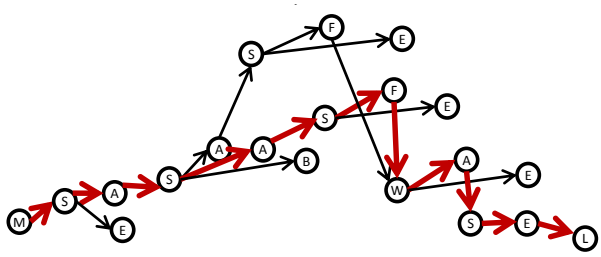

Figure 11: Transaction Graph for transaction in Figure 3. Record labels are omitted for simplicity.

detoured, and assigned a match id of 1. Record 8 marks the begining of the execution of the detoured callback. It calls the actual callback method, `reqCallback`, which has a method id of 19. This method executes between records 9 and 13. We can link records 8 and 9 because they have the same thread id, and will always follow each other (§ 5.3). When `reqCallback` executes, it makes another asynchronous call. This is the call to the UI dispatcher. We detour the callback, and assign it a match id of 2. The actual callback method, of course, is `UpdateUI`, which has the method id of 21.

The completion of this method is indicated by record 16. We note that this method ran on the UI thread. Record 17 indicates that a LayoutUpdated event was triggered immediately after the execution of this method, which means that this method must have updated the UI.

6 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.

6.1 User transactions

We represent user transactions by directed acyclic graphs. The graph is generated from the trace data. Consider the trace in Table 1. It is converted to the graph in Figure 10.

The graph contains five types of nodes, namely: (M) User Manipulation, (S) Upcall start, (E) Upcall end, (A) Async call start, and (L) Layout updated. Each node represents one trace record [4] and is identified by the type and the record id. The mapping between node types M, S, E, A and L and the record types can be gleaned from Table 1.

[4] *CallStart*, *CallEnd* and *CallBackStart* records are used for bookkeeping purposes only, and are not mapped to nodes.

The edges between nodes represent causal relationships. For example, the UIManipulationStarted event $M1$ triggers the start of the handler $S2$. Similarly, the start of callback execution $S9$ was caused by the asynchronous call $A4$. We also say that an upcall start node "causes" any subsequent activity on that upcall. Hence we draw $S2 \rightarrow A4$, as the async call was made during execution of the upcall, and $S2 \rightarrow E6$, 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.

6.2 Critical Path

The critical path is the bottleneck path in the user transaction (§ 3). The basic algorithm for finding the critical path is simple. Consider Figure 10. We traverse the graph backwards, going from the last UI update $(L17)$, to the user manipulation event that signals the start of the transaction $(M1)$, traversing each directed edge in the opposite direction. This path [5], when reversed, yields the critical path: $M1$, $S2$, $A4$, $S9$, $A11$, $S15$, $E16$, $L17$. Even this simple example shows that we correctly account for time spent inside upcalls: for example, the edge $(S9, E13)$ is not on the critical path, which means that any activity in the `reqCallback` method (See Figure 7), 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 § 3, 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 (§ 7), 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 11, which represents the transaction shown in Figure 3: 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

[5] This algorithm always terminates because the transaction graph is always acyclic. Also, we are guaranteed to reach an M node from an L node, with backward traversal. We omit proofs.

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 (§ 9). 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*. These transactions need to be processed differently, since they may not end with UI updates. We omit details due to lack of space. We handle sensor-driven transactions in a similar manner.

6.3 Exception path

When the app crashes, we log the exception information including the stack trace of the thread that crashed (§ 5.6). 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.

6.4 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 § 8.1.3.

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

7 Developer Feedback

The AppInsight server analyzes the collected traces using the methods described in § 6. The developers use a web UI to access the results. Figure 12 shows a collage of some of the views in the UI.

For ease of navigation, the UI groups together identical transactions (§ 6.4) ((a) in Figure 12). 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, the developer can focus her 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.

8 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.

8.1 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

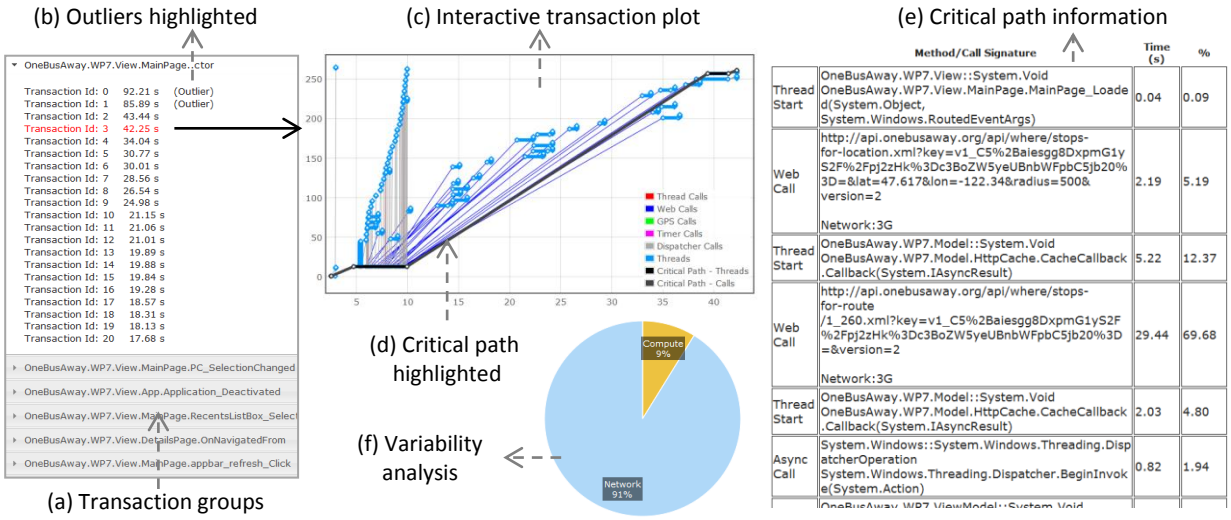

Figure 12: Collage of some of the views in the developer feedback UI.

total num of apps	30
total participants	30
unique hardware models	6
unique hardware+firmware	14
start date	03 April 2012
end date	15 August 2012
total num of app launches (sessions)	6752
total minutes in apps	33,060
total user transactions	167,286
total timer transactions	392,768
total sensor transactions	3587

Table 2: Summary statistics from our deployment

was developed by an author of this paper. 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 plans.

Table 2 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 § 6.2). 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 3 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 13 shows the CDF of the length of user transactions (i.e., the length of their critical path). Only 15% of the transactions last

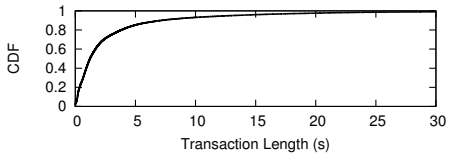

Figure 13: CDF of user-transaction duration.

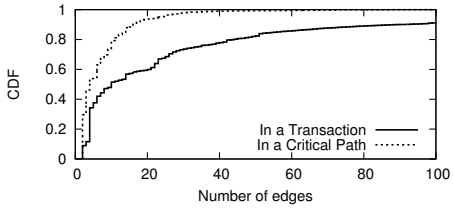

Figure 14: CDF of number of edges in user transactions and in critical paths. The X-axis has been clipped at 100. The top line ends at 347, and the bottom ends at 8,273.

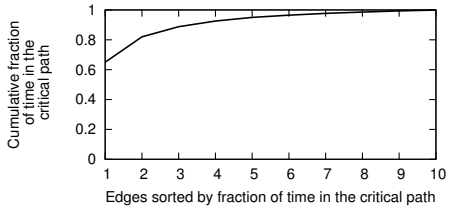

Figure 15: Cumulative fraction of time in the critical path as a function of number of edges.

more than 5 seconds. The app developer would likely want to focus his debugging and optimization efforts on these long-running transactions.

8.1.1 *User Transactions and Critical Paths*

In Table 3, we see that the average number of asynchronous calls per user transaction varies from 1.2 to 18.6

App description	# Users	# Sessions	Avg session length (s)	#User trans--actions	#Async calls	Avg #parallel threads	#trans inter--rupted	Perf overhead ms/trans	Perf overhead ms/s	Network overhead b/trans	Extra data transfer
News aggregator	22	604	88	11738	42826	5.50	1732	3.02	0.69	311	3.2%
Weather	25	533	31	4692	8106	1.92	541	0.31	0.09	162	2.9%
Stock information	17	460	32	4533	5620	1.00	486	0.20	0.06	91	8.6%
Social networking	22	1380	622	48441	900802	7.60	6782	3.48	0.21	487	8.0%
Multiplayer game	21	1762	376	68757	359006	2.28	719	0.18	0.26	27	79.0%
Transit info	7	310	37	1945	40448	4.88	182	2.96	0.85	355	0.9%
Group discounts	9	67	306	1197	3040	6.62	109	0.99	0.06	212	2.3%
Movie reviews	7	48	394	1083	7305	6.56	80	0.51	0.08	97	2.7%
Gas station prices	8	110	48	1434	2085	2.11	72	0.14	0.04	91	1.9%
Online shopping	14	43	512	1705	25701	2.74	349	0.18	0.06	24	4.7%
Microblogging	3	333	60	3913	19853	2.02	386	0.89	0.28	181	2.2%
Newspaper	10	524	142	13281	24571	4.85	662	0.33	0.06	92	1.2%
Ticket service	7	64	530	171	9593	3.70	38	0.05	0.57	9	2.9%

Table 3: 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 § 8.3.1.

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 14 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 15. 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.

8.1.2 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.

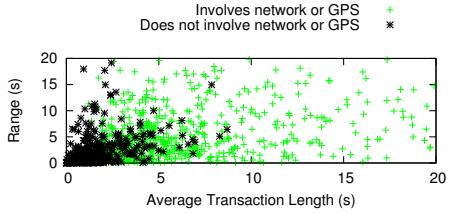

Figure 16: Variation in transaction length for each group. Both axes are clipped at 20.

8.1.3 Aggregate Analysis

We analyzed the trace data from our deployment using techniques described in § 6.4. For the data in Table 2, 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 16 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 (§ 6.4). 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 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 (§ 6.4). 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.

8.2 Case Studies

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

8.2.1 App 1

One of the apps in our deployment was developed by an author of this paper (see § 8.1). AppInsight feedback helped the author 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. The developer had occasionally received crash logs from the Windows Phone developer portal, but logs contained only the stack trace of the thread that crashed. While the developer knew that a routine that split a line into words was crashing, there was not enough information for the developer to diagnose the failure. When the app was instrumented with AppInsight, the developer received the entire exception path. This included the web call and the URL from where the line was fetched. The developer replayed the URL in his app in a controlled setting, and discovered that his 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). The developer spotted that only the user transactions at the start of user sessions experienced these abnormal latencies. He 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. He redesigned his code to force-load the DLLs earlier.

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

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

8.2.2 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.

8.2.3 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.

8.3 Micro-benchmarks

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

8.3.1 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. The buffer is flushed to disk when full[6] or when the app exits.

[6]We use a two-stage buffer to prevent data loss during flushing.

Log Method	Overhead (μs)
LogUpcallStart	6
LogUpcallEnd	6
LogCallStart	6
LogCallEnd	6
LogCallbackStart	6
LogAsyncStart	12
LogObject	12
LogParameters	60

Table 4: Overhead of AppInsight Logger. Averaged over 1 million runs, on a commonly used phone model.

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. The maximum overhead per user transaction is 30ms (average 0.57ms). Since most transactions are several seconds long (see Figure 13), 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 3 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%.

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 3 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) 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).

8.3.2 Coverage

AppInsight uses several heuristics (see § 5) 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, 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[7]. 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.

9 Discussion

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 [7], 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. Third, AppInsight cannot untangle complex dependencies introduced by counting semaphores. The Silverlight framework for Windows Phone [15] does not currently support counting semaphores. 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 § 3 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 (§ 6.2). It is up to the developer to determine which of these paths are important to investigate. Despite these limitations, results in § 8 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.

Applicability to other platforms: The current implementation of AppInsight works for the Windows Phone platform. However, the core ideas behind AppInsight can be applied to any platform that has certain basic characteristics. First, the applications need to have a single, dedicated UI thread. Second, we need the ability to rewrite byte code. Third, we need the ability to correctly identify all possible upcalls (i.e., calls into the user code by the system) and thread start events triggered by the UI itself. Fourth, the system needs to have a set of well-defined thread synchronization primitives. These requirements are not onerous. Thus we believe that AppInsight can be ported to other mobile platforms as well, although we have not done so.

10 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 [12] 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 [4] 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

[7] Some apps require non-random input at the beginning.

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 [9] and Pinpoint [5] 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. [2] 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 [19] on finding the critical path in the execution history of parallel and distributed programs. More recently, Barford and Crovella [3] 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 [8] and PreEmptive [16] 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 [10], it does not necessarily help with performance analysis [1]. Several researchers [17, 14] 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.

11 Conclusion

AppInsight helps developers of mobile apps monitor and diagnose the performance of 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.

Acknowledgments

We thank Ronnie Chaiken and Gleb Kirosheev for discussions and support during AppInsight development. We also thank Petros Maniatis and the anonymous reviewers for their comments on earlier drafts of this paper.

References

[1] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. There's an app for that, but it doesn't work. Diagnosing Mobile Applications in the Wild. In *HotNets*, 2010.

[2] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performane Debugging for Distributed System of Black Boxes. In *SOSP*, 2003.

[3] P. Barford and M. Crovella. Critical Path Analysis of TCP Transactions. In *ACM SIGCOMM*, 2000.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[5] M. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Mangement. In *NSDI*, 2004.

[6] J. Elliott, R. Eckstein, M. Loy, D. Wood, and B. Cole. *Java Swing, Second Edition*. O'Reilly, 2003.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Seth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[8] Flurry. `http://www.flurry.com/`.

[9] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP*, 2009.

[11] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Usenix Windows NT Symposium*, 1999.

[12] M. Jovic, A. Adamoli, and M. Hauswirth. Catch Me if you can: Performance Bug Detection in the Wild. In *OOPSLA*, 2011.

[13] S. Lidin. *Inside Microsoft .NET IL Assembler* . Microsoft Press, 2002.

[14] A. Pathak, Y. C. Hu, and M. Zhang. Where Is The Energy Spent Inside My App? Fine Grained Energy Accounting on Smartphones with Eprof. In *EuroSys*, 2012.

[15] C. Perzold. *Microsoft Silverlight Edition: Programming Windows Phone 7*. Microsoft Press, 2010.

[16] Preemptive. `http://www.preemptive.com/`.

[17] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-Layer Approach. In *MobiSys*, 2011.

[18] A. Whitechapel. *Windows Phone 7 Development Internals*. Microsoft Press, 2012.

[19] C.-Q. Yang and B. P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *IEEE DCS*, 1988.