Appstrument - A Unified App Instrumentation and Automated Playback Framework for Testing Mobile Applications

Vikrant Nandakumar(✉), Vijay Ekambaram, and Vivek Sharma

IBM Research - India, Bangalore, India
{vikrant.nandakumar,vijaye12,vivek.irl}@in.ibm.com

Abstract. Mobile Test Automation is gaining significant importance for an app-tester because it helps to alleviate the voluminous effort and time associated in thoroughly testing an application. Challenges like diversity in mobile hardware, multiple operating systems, ever-increasing application complexity and high volume of test cases etc. reiterate the importance of exploiting automation techniques for mobile application testing. In order to exhaustively capture user actions during the record-phase, faithfully reproduce those actions during playback-phase and also to capture the relevant metrics while playing back, instrumentation of the *Application-under-test* (AUT) becomes an imperative process. However, the type and level of instrumentation is different and is very specific to the category of testing which has to be automated. This paper presents ***Appstrument***, a unified framework for instrumenting mobile applications to make them ready for functional, performance and accessibility testing. This framework allows instrumenting the application to get it ready for either a single category of testing or a combination of two or more of these categories, with multiple optional features for each category. In addition to this, given a test script, the framework also supports automated playback of instrumented applications. *Appstrument* has been deployed and tested against some popular applications from *Google Play* (Android apps) and some IBM in-house iOS applications. Results indicate that this framework is able to successfully instrument a sizeable number of applications and effectively playback user-defined test cases automatically to collect relevant metrics/results corresponding to each category of testing.

Keywords: Mobile applications · Instrumentation · Android · iOS · Testing · Functional · Performance · Accessibility

1 Introduction

Testing mobile applications is an emerging research area that faces a variety of challenges due to increasing number of applications getting developed and a plethora of new devices being released into the market. These new devices have varied form factors, screen size, resolution, OS, hardware specification etc.

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2014
I. Stojmenovic et al. (Eds.): MOBIQUITOUS 2013, LNICST 131, pp. 474–486, 2014.
DOI: 10.1007/978-3-319-11569-6_37

which increases the difficulty to effectively test an application. Also, in comparison to conventional Desktop and Web applications, mobile applications have shorter release cycles (lesser time-to-market) and the update frequency is high, making it necessary for the tester to perform additional testing quite often. Due to these factors, testing a mobile application becomes a very expensive, laborious and time consuming process. IDC [1] predicts that smartphone shipments will reach 978 million in 2014. Hence, with the emerging smartphone market, there is a lot of emphasis on how applications perform on the actual devices. Of late, speed and performance have become the primary concern for consumers of mobile applications. This fact is emphasized in a recent Compuware survey which found that 4 out of 5 users expected an app to launch in 3 s or less. Considering these factors, its imperative to have an automated testing solution which would help save time, minimize human effort and ensure better quality and performance of mobile applications. In order to exhaustively capture user actions during the record-phase, faithfully reproduce those actions during playback-phase and also to capture the relevant metrics while playing back, instrumentation of the *Application-under-test* (AUT) becomes an imperative process. Instrumentation is the process of inserting few lines of code (hooks) in the original AUT to facilitate effective recording and automated replay of test cases.

In this paper, we present *Appstrument*, a unified app instrumentation and automated playback framework for testing mobile applications. *Appstrument* supports functional testing, performance testing and accessibility testing on Android and iOS applications. For each category of testing, this framework supports multiple features which can be used in isolation or in combination with a feature from a different category. Furthermore, the framework also supports automated playback of instrumented applications.

The rest of the paper is organized as follows. Section 2 talks about the motivation for this research work. The detailed architecture, design and implementation is discussed in Sect. 3. The framework evaluation and results are presented in Sect. 4. Section 5 throws light on a few other similar efforts in this domain, followed by Future work and Conclusion in Sect. 6.

2 Motivation

Mobile App instrumentation is primarily needed for getting complete control over an application during execution. Instrumentation is becoming necessary in automated mobile application testing due to the stringent nature of the mobile OS. Most of the existing solutions which bypass the app instrumentation requirement aren't straight forward and requires the device to be rooted/jail-broken [2]. Some of the testing solutions provided by the OSes like Android, suffer with the disadvantage of requiring the testers to code up the test case in a formal language like Java, VBScript etc. Furthermore, the current instrumentation and playback techniques used in testing mobile apps are very purpose-specific and limited to a particular type of testing. Lack of a unified consolidated framework to perform automated mobile application testing is the motivation behind this research work.

3 *Appstrument* Framework for Instrumentation and Playback

This section describes about the detailed design, architecture, implementation and working of the *Appstrument* framework.

3.1 System Architecture and Implementation

The overall architecture of the *Appstrument* framework is depicted in Fig. 1. It comprises of five different components

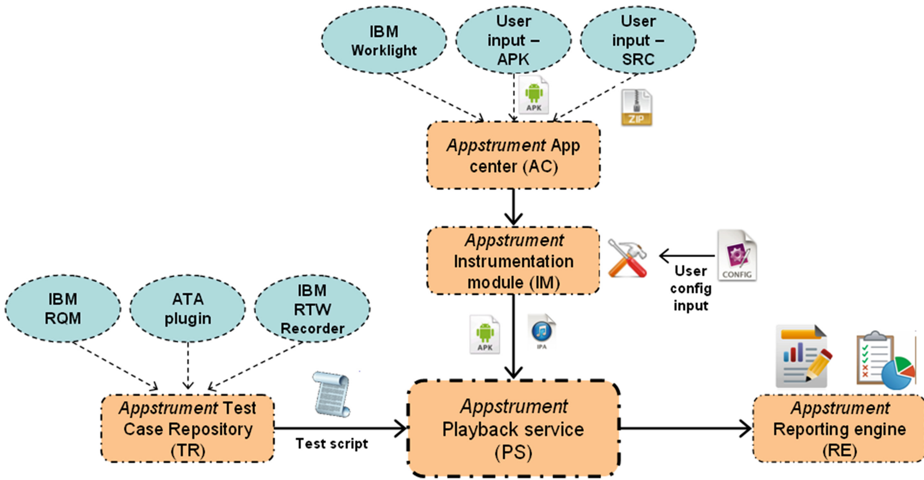


Fig. 1. *Appstrument* - Overall Architecture

1. ***Appstrument* Test Case Repository (TR)** is a collection of automated test scripts (list of test steps) which can be executed on the AUT by the playback service. TR is integrated with three different plugins viz. `IBM Rational Quality Manager` (RQM) [9], Automating Test Automation [3] and `IBM Rational Test Workbench` [10].
   (a) `IBM Rational Quality Manager` offers a collaborative environment for test planning, construction, and execution. It acts as a centralized repository for test cases and is integrated with a number of popular testing products available in the market. *Appstrument* TR has integration with RQM via an adapter through which it can fetch the test cases and push them to the Playback Service for execution on the AUT.
   (b) Automating Test Automation (ATA) [3] is a tool which addresses the problem of creating automated test scripts from manually written test cases. ATA integrates with the *Appstrument* TR and stores the tuple-based English-like Clearscript statements obtained by converting manual test cases.

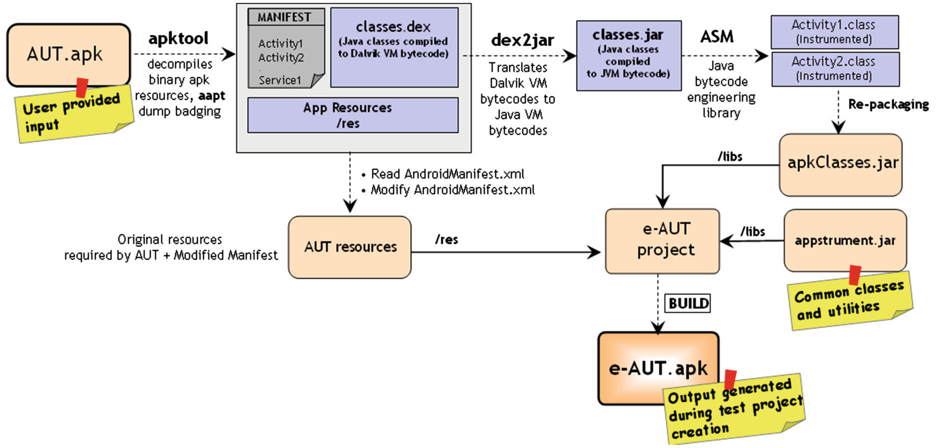


Fig. 2. Instrumentation in Android

   (c) `IBM Rational Test Workbench` consists of a recorder engine which can capture the user-actions in the form of Clearscript statements. The tester can go with this option in situations when there are no manually written test cases available for the AUT.
2. ***Appstrument* App Center (AC)** contains the applications which are available for *Appstrument* Instrumentation Module (IM) for instrumentation.
   (a) `IBM Worklight` [11] is a comprehensive mobile application platform to build, run and manage mobile applications. The `IBM Worklight Application Center` [12] is an enterprise application store which allows the enterprise administrator to install, configure and administer a repository of mobile applications for use by individual employees and groups across the enterprise. *Appstrument* AC module has an in-built adapter to pull an application directly from the Worklight Application Center and pass it on to the *Appstrument* IM module.
   (b) *Appstrument* AC also provides an interface for the user to upload an APK file or the application source code in a zip file format. The need of an APK binary file is primarily for Android applications. However, in the case of iOS, there is a requirement for the application source code to be uploaded to the AC.
3. ***Appstrument* Instrumentation Module (IM)**
   (a) **Instrumentation in Android:** In the case of Android, the AUT is decompiled, reverse engineered and instrumented. The detailed instrumentation process in Android is described in Fig. 2. The following are the steps involved in the Android instrumentation process

- The Android application package referred to as AUT.apk file is given as input by the tester.
- Using the `dump badging` command of the Android Asset Packaging Tool (*aapt*) [4] tool, high level information about the .apk file like package name, versionCode, versionName, permissions, features, minSdkVersion, targetSdkVersion, app label, app icon file path, main launchable activity name etc. are obtained.
- The AUT .apk file is reverse engineered using the Android *apktool* [5]. This tool unzips the .apk file generating all the resources (images, layout XMLs, string resource XMLs, manifest file, etc.) and the compiled classes (*classes.dex*) of the AUT. *classes.dex* is nothing but byte-code of all AUT classes compiled for the Dalvik virtual machine. The Manifest XML is slightly modified to set internal flags for the *Appstrument* PS to automatically detect that the AUT is instrumented and ready for playback.
- In the next step, dex2jar [6] tool is used to decompile *classes.dex* to decode and understand the underlying Java classes. This tool translates the Dalvik VM byte-codes to the Java VM byte-codes. The resultant file is *classes.jar* which contains corresponding Java classes from *classes.dex*.
- The individual files from *classes.jar* is extracted by using the Java jar tool. From the Manifest file, the declared Activity information is obtained and using the ASM byte-code engineering library [7], the Java byte-code is loaded into the memory and altered in different ways.
- Once the bytecode is loaded, for each Activity class declared in the manifest, modifications are performed on the Android activity life cycle methods viz. *onCreate()*, *onStart()*, *onResume()*, *onPause()* and *onDestroy()*. These methods are modified to include an additional line towards the end of each method, which enables a call-out to *Appstrument* instrumenter class, thereby establishing control over the complete application during runtime.

(b) **Instrumentation in iOS:** In the case of an iOS application, there is a requirement for the AUT's source code to carry out the instrumentation process. The *Appstrument* library is linked with the AUT's source code and the AUT is built to get the instrumented AUT. However, in this process there is no modification of source code of the original AUT. We make use of the `Objective-C` *Method Swizzling* [8] concept to make the method in the AUT call an alternate implementation which is different from the original implementation. In the below example, since directly overriding the *sendEvent* method (called by UIApplication to dispatch events to views inside the window) would break the responder chain sequence, the `Objective-C` *Method Swizzling* concept is used to hook into our *Appstrument* adapter code on the iOS event bus using the original *sendEvent* method. Once the *sendEvent* method is swizzled, *Appstrument* gains access into the event bus at runtime and can push a new event during playback of the AUT.

```
<?xml version="1.0"?>
<Appstrument>
  - <FunctionalTesting>
        true
        <ImageCaptureAndMarkup>true</ImageCaptureAndMarkup>
        <VerificationPoints>true</VerificationPoints>
        <DynamicFind>true</DynamicFind>
    </FunctionalTesting>
  - <PerformanceTesting>
        false
        <ResourceMonitoring>false</ResourceMonitoring>
        <MethodLevelProfiling>false</MethodLevelProfiling>
        <EventProfiling>false</EventProfiling>
        <KernelAndUIthreadMapping>false</KernelAndUIthreadMapping>
    </PerformanceTesting>
  - <AccessibilityTesting>
        false
        <ImageCaptureAndMarkup>false</ImageCaptureAndMarkup>
        <AccessibilityAutoCorrection>false</AccessibilityAutoCorrection>
    </AccessibilityTesting>
</Appstrument>
```

Fig. 3. *Appstrument* config file

```
− (void)sendEvent : (UIEvent∗)event;
```

(c) **Appstrument config file:** The *Appstrument* IM module takes a *Appstrument config* file as input before it starts instrumenting the AUT. This *Appstrument config* file defines the different configuration parameters which have to be considered for generating the instrumented AUT. This file is in XML format and has a boolean flag *(true/false)* defined against each of the features for every category of testing. These features are optional for the user to choose and are in addition to (i) basic playback for Functional Testing, (ii) playback + resource monitoring for Performance Testing, (iii) playback + compliance check for Accessibility Testing. The details about the various features are explained in the following section. Figure 3 shows a sample *Appstrument config* file.

4. ***Appstrument* Playback Service (PS)**
The Playback Service is responsible for the execution of a test script on the AUT. PS picks up the test script from the Test case Repository (TR) and plays it back on the instrumented application from the Instrumentation Module (IM). The output of the playback is sent to the Reporting Engine (RE). Depending on the *Appstrument config* file provided to the IM module, certain features are enabled or disabled for each category of testing during playback. The details of features which are supported by the PS are as follows:
   (a) **Features for Functional Testing**
      i. **Image Capture and Mark-up** - By enabling this feature during playback, the Playback Service will capture the snapshot of AUT's screen on which the action happens for a particular test step. Additionally, on the captured snapshot, the PS will draw a bounding box around the UI element which is associated with that test step to indicate to the tester where exactly in that screen the action has happened. For example, if a user clicks on the "Submit" button on a login page, then the snapshot of the login page with a bounding box around the "Submit" button is provided in the *Appstrument* Report.

   ii. **Verification Points** - This is a feature by which the Playback Service can check if a particular UI element conforms with a tester-specified property like (i) verifying if a TextView contains the text as defined by the tester, (ii) location of the TextView on the screen etc. The verification point definitions are included as a part of the test script to be played back on a AUT.
   iii. **Dynamic Find** - This feature helps the Playback Service to resolve dis-ambiguities in the test case to ensure a faithful playback on the AUT. For example, if the test step is "*Click on Submit button to the left of Reset Button*", the Playback Service should be aware of other UI elements in close proximity to the *Submit* button. The details about these UI elements, referred as *associated element description*, are required to execute a test step with greater accuracy.

(b) **Features for Performance Testing**
   i. **Resource Monitoring** - This default Performance Testing feature captures the various parameters like CPU Utilization, RAM Usage, Virtual Memory Usage, Network Traffic (incoming & outgoing) etc. *Appstrument* PS captures these parameters without any requirement for instrumentation of AUT.
   ii. **Method-level profiling** - Enabling this feature would capture all the method level details with respect to the AUT. *Appstrument* logs messages during method start and exit to track these information.
   iii. **Event profiling** - This feature captures all the system, user, UI events of the AUT occurring during playback to visualize the control and the data flow of the application.
   iv. **Kernel and UI thread Mapping:** Having this feature enabled would help in capturing all the kernel thread activities (like Wifi service calls, Activity Manager, Window manager, etc.) and map these activities with respect to the AUT events to understand the performance bottlenecks.

(c) **Features for Accessibility Testing**
   i. **Image Capture and Mark-up** - This feature is very similar to the Image Capture and Mark-up feature mentioned in Functional Testing. Unlike Functional Testing, the principal difference in accessibility testing is, instead of drawing a bounding box on the target UI element (for a single test step), the *accessibility non-compliant* UI element is highlighted to the tester in the playback report.
   ii. **Accessibility Auto Correction** - Enabling this feature would not only identify the *accessibility non-compliant* UI element, but also try to perform an auto correction to make it compliant. For example, for an Android EditText widget, if the `android:hint` property is not set, this feature would employ the Dynamic Find (Functional testing feature) to identify a nearby associated label to fill in the property field thereby auto correcting the non-compliance error.

Appstrument Playback Mini Report: sample_test [7/22/13 12:57 PM]

Test:	sample_test [executed on 7/22/13 12:53 PM]
Categories to test:	Functional, Performance, Accessibility
Application:	IBM NUC Sensing App (version 4; created on 7/22/13 12:50 PM; executed on 7/22/13 12:53 PM)
Execution Status:	**Functional** – All steps passed **Performance** – Passed **Accessibility** - Failed. 1 or more elements failed the test
Device:	samsung GT-P3100 (Android; API level 15)

Fig. 4. *Appstrument* Playback Mini Report (Sample)

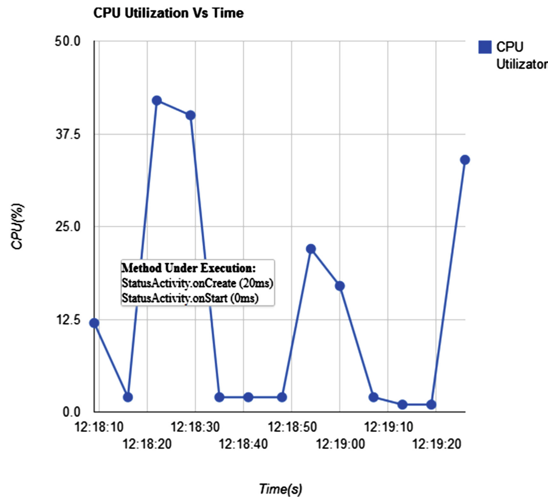


Fig. 5. *Appstrument* Performance Testing Report (Sample)

5. ***Appstrument* Reporting Engine (RE)**
Reporting Engine (RE) generates test reports for the test script executions on the AUT by the Playback Service. Depending upon the flags set in the *Appstrument config* file, reports are generated for various categories of testing along with the optional features under each category. RE generates two types of reports. Figure 4 shows an *Appstrument* Playback Mini Report which gives a summary of the test execution using the *Appstrument* framework. Figure 5 shows a detailed Performance Testing Report depicting the CPU utilization of the AUT with respect to time. Upon hovering over the graph, the method level details executed at that particular instance of time is also shown. Similarly, *Appstrument* RE generates detailed category-wise/feature-wise reports for a particular test script execution.

4 Evaluation

4.1 Testing Environment

In-order to show the impact of dynamic feature selection, we tested the *Appstrument* with the following categories of application:

- **Android-Native Application:** Apps considered for testing in this category are Calculator, S Planner, Quickoffice, IBM Sametime, Facebook and IBM sensing app.
- **Android-Hybrid Application:** Apps considered for testing in this category are RedBus, Twitter, BookMyShow, IRCTC Online Booking and IBM Worklight apps.
- **iOS-Native Application:** Since iOS instrumentation requires source-code for instrumentation, we used 5 internal IBM iOS native apps. (names not mentioned to maintain confidentiality)
- **iOS-Hybrid Application:** For the same reason stated above, we used 5 internal iOS hybrid IBM Worklight apps.

Each application is tested with 5 different test cases to ensure that most features of the AUT are covered. We used Samsung Galaxy Tab (Android 4.0) and iPhone 4S (iOS 6.1) as target devices and the *Appstrument* framework is deployed on a Lenovo T430 machine running Win 7 with 4 GB RAM and Intel Core i7 processor.

4.2 Time to Instrument

Time to Instrument refers to the total time *Appstrument* has taken to instrument the AUT. For evaluation purposes, we have instrumented all the applications mentioned in the Sect. 4.1. In Table 1, we have shown the time *Appstrument* has taken to instrument few popular Android applications for functional testing with all 3 features included. Instrumentation of the applications is a one-time process, as *Appstrument* saves the instrumented AUT in its repository for future testing purposes.

Table 1. Time to Instrument

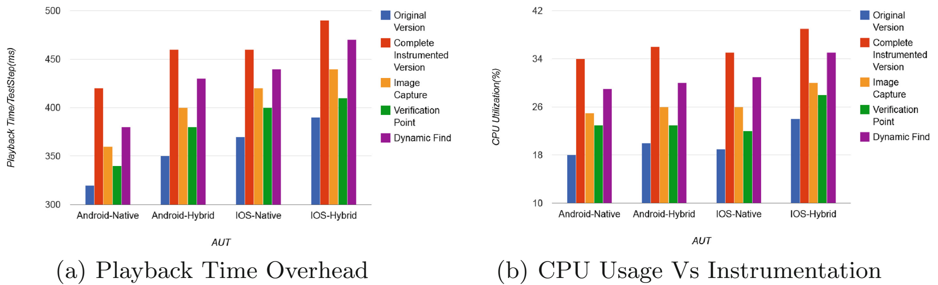
AUT	Time to Instrument(ms)	Binary size(MB)
`IBM SameTime`	47280	1.13
`Whatsapp`	205237	10.4
`Facebook`	402860	13.7
`Skype`	246991	15.1
`Flipboard`	58499	2.3

4.3 Record and Playback

In order to evaluate *Appstrument* with respect to its playback capability, we tested each AUT (mentioned in the Sect. 4.1) with varying test-cases (both deterministic and non-deterministic). Non-Deterministic test-cases refer to the test-cases where the flow of execution and the interval time between test-steps varies with respect to the run-time inputs. *Appstrument* is able to playback all the test-cases with reduced overhead. Analysis with respect to the overhead is mentioned in the following sections.

4.4 Impact of Instrumentation on Functional Testing

Appstrument provides the various instrumented versions (as explained in Sect. 3.1) with respect to functional testing. A tester can choose the features required and the instrumentation overhead varies depending on these set of features.



(a) Playback Time Overhead (b) CPU Usage Vs Instrumentation

Fig. 6. Impact of instrumentation on functional testing

Figure 6(a) compares and analyzes the playback time of various instrumented versions. X axis represents the time taken for each test-step to execute during playback. From the graph, we observed the playback time of the various featured instrumented versions follow this order: {Complete instrumented version > Dynamic Find instrumented version > Image Capture instrumented version > Verification point instrumented version > Original Version} (Note: Original Version refers to the un-instrumented AUT). Interestingly, further analysis shows that the CPU utilization of these instrumented versions in Fig. 6(b) follows a similar pattern as the playback time. Since *Appstrument* allows specific configurations of the required features for functional testing, testers can reduce the instrumentation overhead based on their requirements. Table 2 shows the percentage of overhead reduced (with respect to the playback time) for various partially instrumented versions as compared to the complete instrumentation.

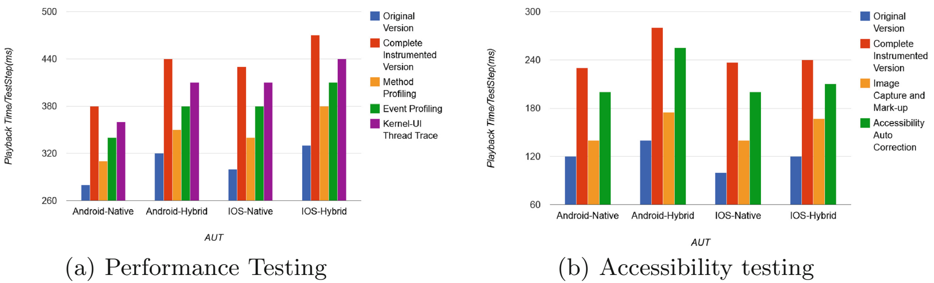
4.5 Impact of Instrumentation on Performance Testing

In Fig. 7(a), we compare various instrumented versions (as mentioned in Sect. 3.1) with respect to performance testing. Playback time of these instrumented versions varies with respect to the features selected. Table 3 shows the performance

Table 2. Functional testing-overhead reduction

AUT category	Image capture	Verification point	Dynamic find
`Android-Native`	14 %	19 %	10 %
`Android-Hybrid`	3 %	17 %	7 %
`iOS-Native`	9 %	13 %	4 %
`iOS-Hybrid`	10 %	16 %	4 %

overhead reduction obtained with respect to selective instrumentation as compared to complete instrumentation. For testers, there is a huge scope to reduce instrumentation overhead by choosing appropriate configurations based on their requirements.



(a) Performance Testing (b) Accessibility testing

Fig. 7. Impact of instrumentation on performance and accessibility testing

Table 3. Performance testing-overhead reduction

AUT category	Method profiling	Event profiling	Kernel-UI thread mapping
`Android-Native`	18 %	11 %	5 %
`Android-Hybrid`	20 %	14 %	7 %
`iOS-Native`	21 %	12 %	5 %
`iOS-Hybrid`	19 %	13 %	6 %

4.6 Impact of Instrumentation on Accessibility Testing

Figure 7(b) shows the analysis of the selective feature capability of *Appstrument* with respect to the Accessibility testing. Since features like Accessibility auto-correction and Image Capture and Mark-up (as mentioned in Sect. 3.1) incur more overhead as compared to the basic accessibility testing, the tester can choose these features as and when they are required. From Fig. 7(b), we also

understand that the Accessibility auto-correction feature has a heavy overload on the AUT's performance as it nearly doubles the playback time of the complete test-case.

5 Related Work

A cursory examination of the literature in test automation for mobile applications clearly throws light on the significant attention devoted to the instrumentation techniques. Recent mobile testing publications like [14,16,18,19] clearly indicate the necessity for code instrumentation in mobile testing. Mobile automation testing products in the market like `TestStudio` from Telerik, `MonkeyTalk` from Gorilla Logic and `Jamo` from Jamo Solutions do instrumentation for mobile test automation. Recent papers in mobile performance testing like Energy Profiler [18] and AppInsight [14] instruments the AUT to capture all the performance related metrics with respect to the AUT. SIF [15] talks about creating abstractions for selective instrumentation and also recounts the advantages of selective instrumentation. Comparing *Appstrument* with these prior-arts, *Appstrument* provides a much detailed architecture for complete as well as selective instrumentation with respect to most common mobile testing categories like Functional, Performance and Accessibility testing.

6 Future Work and Conclusion

In this paper, we presented *Appstrument* - a unified framework for mobile application instrumentation and playback for automated testing. In future, we envision to include several other categories of testing like security testing, usability testing, localization testing etc. We have plans to expand on the features supported under each umbrella of testing. We intend to drastically improve our coverage with the diversity of applications and devices the framework currently supports. Efforts are on to come up with a solution to get rid of the source code requirement for iOS instrumentation. In sum, we foresee a better, powerful, efficient and versatile *Appstrument* framework in future which overcomes the various challenges associated with mobile test automation.

References

1. IDC Worldwide Smartphone 2012–2016 Forecast Update, June 2012
2. Gomez, L., et al.: RERAN: timing-and touch-sensitive record and replay for Android. In: ICSE '13 (2013)
3. Thummalapenta, S., et al.: Automating test automation. IBM Research report (2011)
4. Android, AAPT tool, July 2013. http://elinux.org/Android_aapt
5. Android, apktool, July 2013. https://code.google.com/p/android-apktool/
6. Dex2Jar tool, July 2013. http://code.google.com/p/dex2jar/
7. Bruneton, E.: ASM 4.0 - A Java bytecode engineering library, OW2 Consortium

8. Cocoa Dev, Method Swizzling, July 2013. http://cocoadev.com/MethodSwizzling
9. IBM Rational Quality Manager, July 2013. http://www-03.ibm.com/software/products/us/en/ratiqualmana/
10. IBM Rational Test Workbench, July 2013. http://www-03.ibm.com/software/products/us/en/rtw
11. IBM Worklight, July 2013. http://www-03.ibm.com/software/products/us/en/worklight/
12. Irvine, M., Maddocks, J.: Enabling Mobile Apps with IBM Worklight Application Center, IBM Redpaper, June 2013
13. Mahmud, J., Lau, T.: Lowering the barriers to website testing with CoTester. In: Proceedings of the 14th International Conference on Intelligent User Interfaces, pp. 169–178 (2010)
14. Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., Shayandeh, S.: AppInsight: mobile app performance monitoring in the wild. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12), pp. 107–120 (2012)
15. Hao, S., Li, D., Halfond, W.G.J., Govindan, R.: SIF: a selective instrumentation framework for mobile applications. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13), pp. 167–180 (2013)
16. Sama, M., Harty, J.: Using code instrumentation to enhance testing on J2ME: a lesson learned with JInjector. In: Proceedings of the 10th Workshop on Mobile Computing Systems and Applications (HotMobile '09) (2009)
17. Zhou, W., Zhang, X., Jiang, X.: AppInk: watermarking android apps for repackaging deterrence. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13), pp. 1–12 (2013)
18. Pathak, A., Hu, Y.C., Zhang, M.: Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof. In: Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12), pp. 29–42 (2012)
19. Nath, S., Lin, F.X., Ravindranath, L., Padhye, J.: SmartAds: bringing contextual ads to mobile apps. In: Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Aervices (MobiSys '13), pp. 111–124 (2013)