Mosaic: Cross-Platform User-Interaction Record and Replay for the Fragmented Android Ecosystem

Matthew Halpern Yuhao Zhu
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
{matthalp, yzhu}@utexas.edu

Ramesh Peri
Software and Services Group
Intel Corporation
ramesh.v.peri@intel.com

Vijay Janapa Reddi
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
vj@ece.utexas.edu

***Abstract*—In contrast to traditional computing systems, such as desktops and servers, that are programmed to perform "compute-bound" and "run-to-completion" tasks, mobile applications are designed for user interactivity. Factoring user interactivity into computer system design and evaluation is important, yet possesses many challenges. In particular, systematically studying interactive mobile applications across the diverse set of mobile devices available today is difficult due to the mobile device fragmentation problem. At the time of writing, there are 18,796 distinct Android mobile devices on the market and will only continue to increase in the future. Differences in screen sizes, resolutions and operating systems impose different interactivity requirements, making it difficult to uniformly study these systems.**

We present Mosaic, a cross-platform, timing-accurate record and replay tool for Android-based mobile devices. Mosaic overcomes device fragmentation through a novel virtual screen abstraction. User interactions are translated from a physical device into a platform-agnostic intermediate representation before translation to a target system. The intermediate representation is human-readable, which allows Mosaic users to modify previously recorded traces or even synthesize their own user interactive sessions from scratch. We demonstrate that Mosaic allows user interaction traces to be recorded on emulators, smartphones, tablets, and development boards and replayed on other devices. Using Mosaic we were able to replay 45 different Google Play applications across multiple devices, and also show that we can perform cross-platform performance comparisons between two different processors under identical user interactions.

I. Introduction

Unlike traditional "run-to-completion" workloads found in desktops and servers, interactive mobile applications terminate at the discretion of the end-user. User interaction serves as the primary input source for mobile applications, where computational activity is generated in response to the timing, type and location of user input. Together, the sequence of user interactions that occur throughout an application use case form an input set for that application. Different user interaction sequences invoke different application behaviors just as the corresponding input sets from traditional computer benchmarking workloads (e.g. SPEC) do during execution.

The impact of user interaction on mobile application behavior requires that developers and researchers evaluate mobile systems under realistic usage scenarios. Interactive mobile applications are designed to directly interact with the end-user through a touchscreen display. Different applications require the user to interact with the application in different ways. For example, first-person shooter games use finger tapping to fire ammunition and navigation services provide "pinch-to-zoom" in order to investigate interesting geographical features. Whether it be taps, pinches, flicks or swipes, each application has its own set of requirements for how user interactions must be performed to provide the intended end-user functionality.

From a program analysis and computer architecture perspective, studying mobile applications requires repeatable user interactivity. That is, the *type*, *timing* and *location* of user interactions must be repeatable across different runs of the application. Techniques that use live human inputs are susceptible to run-to-run variation and do not easily scale. There is a need to be able to accurately reproduce the same execution behavior across several runs of the same application. Interactive mobile applications are time-sensitive and can show measurement variation across users and even individual uses, even when the same functionalities are exercised. To address these issues, several user input replay techniques have emerged; either from a recorded or scripted user interaction sequence.

A key challenge in preserving the type, timing and location of input events across platforms is the mobile device fragmentation problem. Mobile device ecosystems, most notably Android, embrace design freedom that encourages hardware and software diversity across device offerings. At the time of writing, Android has over 18792 distinct device configurations [1]; each with different screen sizes, operating system versions and other hardware and software differences. As we demonstrate, although existing replay tools work out-of-the box for recording and replaying activity on a single device, they do not provide cross-platform portability. User actions from one devices cannot be replayed on other devices.

Cross-platform user input record and replay enables robust application testing and quantitative performance comparisons amongst different mobile devices. While the challenges Android fragmentation imposes on application development are well-known, the same underlying principles also hinder the ability to effectively study mobile device performance, specifically from a user interaction perspective. To evaluate two systems under the same interactive mobile application usage scenario, they require the same interaction sequences that interact with the same user interface (UI) elements under the same timing constraints. However, this is not easy due to the fragmentation issue. In the simplest case, in order to evaluate an identical application found within a smartphone and tablet, screen dimensions require interactions at different coordinates, preventing user inputs from being portable between the two devices. Cross-platform replay is not just a display-level issue. We demonstrate it also requires system-level cooperation between the touchscreen, kernel and application framework.

To overcome these issues, we present Mosaic, a cross-platform user input record and replay tool for Android-based systems. Mosaic virtualizes user interactions in a way that

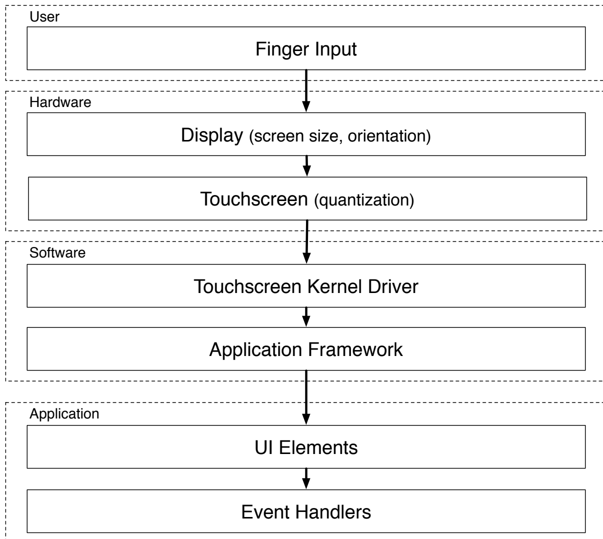


Fig. 1: Android touchscreen input architecture overview.

allows application use cases to replayed across a variety of mobile devices, each with different hardware and software attributes pertaining to fragmentation. In the spirit of the art form from which it derives its name, Mosaic provides the capabilities to cohesively study mobile devices spread across the fragmented Android ecosystem. It does this by mapping a set of touchscreen events from a particular device into a set of virtualized user interactions that can be retargeted and injected into another device. The approach is general enough that it can be used to readily augment existing popular record and replay tools that currently do not support cross-platform portability.

We evaluated Mosaic across a wide range of applications and devices. Our evaluation included several off-the-shelf smartphones, tablets, development platforms and emulators. We were able to successfully run over 45 applications that were downloaded from Google Play across various combinations of hardware and software emulation platforms. All of this was achieved transparently without requiring any source code changes and with almost negligible system overhead.

The rest of the paper is organized as follows. In Sec. II, we introduce the mobile device fragmentation problem. Sec. III presents the design and implementation of Mosaic. Mosaic's cross-platform portability is evaluated in Sec. IV. We demonstrate how Mosaic can be used to conduct a microarchitecture-level performance analysis between an ARM Cortex-A7 and Cortex-A15 processors and discuss over potential use cases in Sec. V. We compare Mosaic with prior work in Sec. VI and conclude the paper in Sec. VII.

II. THE FRAGMENTATION PROBLEM

The Android mobile device ecosystem suffers from a fragmentation issue. To date, Android has over 18792 unique devices on the market [1]. Device manufacturers have the freedom to create mobile devices in any way they wish, leading to both hardware and software diversity, deemed fragmentation. Hardware fragmentation refers to the different displays, processing resources and other components that operate under the operating system version. The most visible example are mobile device form factors (e.g. smartphones, tablets and wearables) that physically look different but run the same Android

TABLE I: Linux Touchscreen Input Events of Interest.

Event	Description
`ABS_MT_SLOT`	Finger Identifier
`ABS_MT_TRACKING_ID`	Event ID
`BTN_TOUCH`	Screen Press/Release
`ABS_MT_POSITION_X`	Finger X-Position
`ABS_MT_POSITION_Y`	Finger Y-Position
`ABS_MT_TOUCH_MAJOR`	Finger Touch Major-axis Length
`ABS_MT_TOUCH_MINOR`	Finger Touch Minor-axis Length
`ABS_MT_WIDTH_MAJOR`	Finger Width Major-axis Length
`ABS_MT_WIDTH_MINOR`	Finger Width Minor-axis Length
`ABS_MT_ORIENTATION`	Finger Orientation
`ABS_MT_TOOL_TYPE`	Touch Tool Type
`ABS_MT_PRESSURE`	Touch Pressure
`ABS_MT_DISTANCE`	Touch Distance
`SYN_MT_REPORT`	End of Multitouch Event Packet
`SYN_REPORT`	End of Event Packet

version. In contrast, software fragmentation relates to software variations across the same device model. For example, different operating system versions and carrier-specific modifications.

The rest of this section describes user interactivity in Android and the corresponding fragmentation issues that concern it. Both hardware and software fragmentation impose challenges to providing accurate, cross-platform user interaction record and replay. Not only do these restrictions stem from differences between device display and touchscreen attributes but also due to kernel- and application framework-level differences. Several other sources of fragmentation exist (e.g. sensors), but we consider them to be beyond the scope of this work since they not directly deal with touch-related inputs.

A. *Android Touch Input Architecture*

A high-level overview of how Android bridges user interactivity to the underlying application code in shown in Fig. 1. The user interacts with application user interface (UI) elements located at various display coordinates. These interactions can range from finger presses, taps and swipes depending on the application functionally the user wishes to invoke. In addition, the user may also exercise multi-finger gestures, such as a pinching or rotation, where multiple fingers concurrently execute these primitives. All the while, the touchscreen tracks finger state atop the display throughout the interaction.

The touchscreen reports user interaction state updates directly to Android's Linux kernel through events, such as those shown in Table I. These events are a subset of the Linux input protocol, mostly from the (multi-)touch specification. Due to space constraints, we only show the events we observed at least once while investigating 26 physical devices, which we will discuss in detail later in the section. The kernel processes touchscreen interactions as event packets sent through the `/dev/input/event` interface. Each event packet begins with a header that consists of an `ABS_MT_TRACKING_ID` event, which is preceded by a `ABS_MT_SLOT` event to uniquely identify a finger when multiple fingers are interacting with the screen. The packet body consists of finger interaction metadata, such as the interaction coordinates and the pressure applied. Each packet ends with a `SYN_REPORT`.

Fig. 2 illustrates an example of a finger press event packet sequence on a tablet. When the finger begins to push down on the touchscreen, the driver creates a new touchscreen event packet by sending `ABS_MT_TRACKING_ID`

with a unique identifier to the kernel. A `BTN_KEY DOWN` event is included because this is the first event packet sent while the finger is touching down on the touchscreen. The finger's current position at $(1152, 486)$ is expressed with `ABS_MT_POSITION_X 1152` and `ABS_MT_POSITION_Y 486`. Fingers make contact with the screen in the form of ellipses. The major (i.e. longest) and minor (i.e. shortest) axes of the ellipse is presented to the system through `ABS_MT_TOUCH_MAJOR 16` and `ABS_MT_TOUCH_MINOR 8`, respectively. `ABS_MT_PRESSURE` corresponds to how hard the finger is pressed against the screen. Finally, the packet sequence ends with a `SYN_REPORT` to signal the system to process the event packet information.

The Android application framework awaits and converts the Linux touchscreen driver events into application-level interactions. The Android `EventHub` service monitors the touchscreen for new events (via `/dev/input/event`) and streams them into a module that determines when enough information is available for the current application's window manager to begin processing a set of user interactions.

The window manager uses the touchscreen event information to identify which application UI element the user is interacting with and then dispatches its corresponding event handler. For example, if the window manager identifies that a button has been pressed, it will call the button's event handler. In Android, this is the `OnTouchListener` handler.

B. Android Hardware Fragmentation

Several hardware aspects, mostly pertaining to screen-related components, restrict user touchscreen interactions from being cross-platform. We present and elaborate on the issues that we identified and addressed while developing Mosaic.

Screen Size and Orientation — Mobile devices with various form factors (e.g. smartphones and tablets) have to support the same set of applications. Form factors are distinguishably different in screen size (e.g. inches, etc) and default orientations (e.g. portrait or landscape). Smartphones have small portrait screens between 2.5" and 5.5" whereas tablets have larger landscape screens at least 9". In between these two extremes are phablets that have 6"-wide screens and much smaller tablets in the 7" to 8" range used in the portrait mode.

Mobile device manufacturers have, and continue to, embrace hardware diversity. To convey the degree of screen-level fragmentation amongst off-the-shelf mobile devices, we data mine `gsmarena.com` [2] for mobile device screen parameters. Across the 1670 different Android-based mobile devices that we found, Fig. 3b shows that their screen sizes are diverse, ranging from the 2.55 inch Sony Ericsson Xperia X10 Mini smartphone to the 13 inch Toshiba Excite 13 tablet. An outstanding majority of the mobile devices in our data set are smartphones that are typically used in portrait mode. However, the distribution of screen sizes are uniformly distributed since the screen size linearly increases as more smartphones are included in the zero to about twelve hundred. The rest of the mobile devices correspond to tablets that are split between the default portrait and landscape modes.

Screen size and orientation fragmentation complicate cross-platform record and replay because they rely on form-factor specific Cartesian coordinate systems. Mobile display coordinate systems originate in the top-left corner of the display in its default orientation. For many applications, devices with

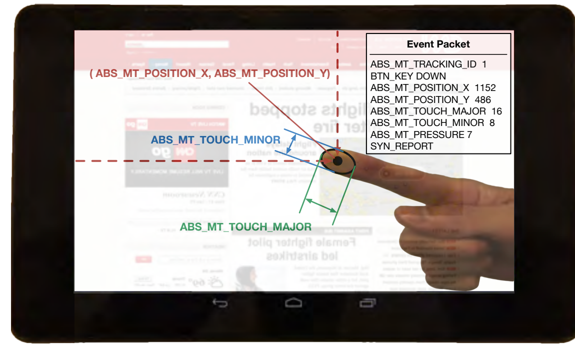


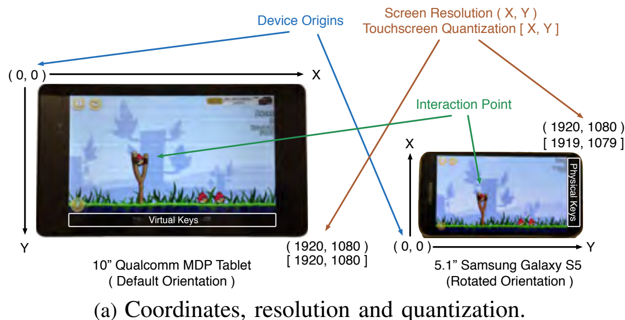
Fig. 2: Example finger press and the event sequence that it generates.

different form factors must have the same display orientation to exercise the same behavior. Fig. 3a shows an example for Angry Birds on a tablet and smartphone. The tablet remains in its default landscape orientation, while the smartphone has to be rotated away from its default portrait orientation to play the game. While the two devices share the same physical screen orientation, their coordinate systems are no longer in sync with one another. The smartphone's origin has moved from the top-left to the bottom-left corner. Mapping user interaction coordinates between the two devices cannot just simply be scaled to work. Pulling back the slingshot on the Qualcomm MDP tablet requires a simultaneous decrease and increase in x and y touchscreen coordinate whereas the rotated Samsung Galaxy S5 origin requires a simultaneous decrease in both the x and y touchscreen coordinates instead.

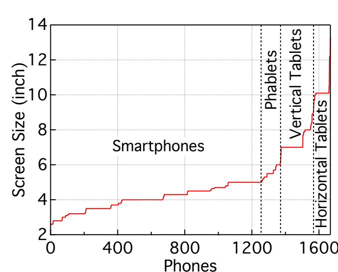
Screen Resolution — While the most visible form of Android fragmentation is screen size, due to diverse consumer demands Android devices are also available in different screen resolutions. Over time, mobile device display technology has improved to provide higher resolution screens. Fig. 3c provides a snapshot of the different of mobile display resolutions on the market based on our mined mobile device display information. Each marker corresponds to a resolution configuration where the x-and y-axes correspond to the pixel widths for the horizontal and vertical dimensions of the display in the mobile device's default orientation. For example, the Qualcomm MDP Tablet and the Samsung Galaxy S5 smartphone in Fig. 3a would be at points $(1920, 1200)$ and $(1080, 1920)$, respectively. The clustered points in the bottom left quadrant of the figure correspond to smartphone display resolutions and the more sparse points in the top left and bottom right quadrants correspond to portrait and landscape tablets, respectively.

Mobile device screen coordinate systems are typically thought of in terms of display pixels. So regardless of how the applications are programmed, UI element shapes and their positioning is ultimately finalized on to the display in the form of pixel occupancy. The same application on two different systems may differ in the number of pixels to place and fill for each of its different UI elements. Applications that follow popular design patterns that embrace relative positioning and sizing, such as grid-, fluid- and responsive-based methodologies, will scale the same across devices, but their UI elements will be located at different pixel points on the screen.

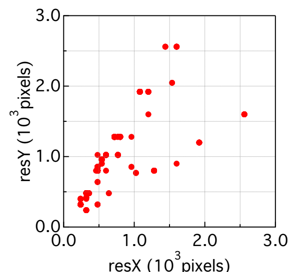
The challenge with deterministically replaying user interactions recorded on one device onto another device with a different resolution is that application UI elements can some-



(a) Coordinates, resolution and quantization.



(b) Screen sizes.



(c) Screen resolutions.

Fig. 3: Understanding mobile device fragmentation, involving screen sizes, coordinates, orientations, quantization, and resolution.

times be pixel-dependent. Simply recording the user interaction locations for one device and mapping those points to another device with a different resolution will not work because the UI elements will be different on the target UI. For example, Pinterest, the visual discovery tool will display more content on tablets than smartphones. We find that application UIs tend to cluster by form factor. As long as devices mapped to one another share (or emulate) similar form factor resolutions, user interactions can be mapped between one another.

Touchscreen Quantization — While the UI is expressed in terms of pixels, user interaction is captured through touchscreen events. Therefore, porting user interactions across devices relies on correctly mapping touchscreen coordinates between the different devices. Naively mapping user interactions at the pixel level does not guarantee portability.

User touchscreen events are generated by converting user finger positions into discrete values that can be processed by the system. The process is known quantization, and the mobile device's touchscreen technology determines the quality and impact of the quantization. In some cases, the screen's quantization matches the screen resolution perfectly, but that is not always the case. Because quantization is not always coupled with resolution, without a clear connection from one device to another, fragmentation at the quantization level posses a major obstacle for cross-platform portability.

Mobile devices do not always have the same touchscreen quantization as resolution. While the Qualcomm MDP tablet and Samsung Galaxy S5 smartphone in Fig. 3a each have quantizations that match their screen resolutions, several of the devices we studied do not. For example, one of the devices has a 1920x180 resolution, but had a 4095x4095 quantization, which is beyond any resolution found in a mobile device today.

C. Android Software Fragmentation

Software fragmentation also impedes cross-platform user interaction record and replay capabilities. Device manufacturers update devices with new drivers and operating system versions that can affect how touchscreen events are handled by the device and also whether or not other device buttons, such as menu buttons, are virtualized on the device.

Touchscreen Driver — The touchscreen driver communicates the touchscreen's state to the operating system. Different touchscreen models communicate state in different ways. In addition, the range of information that a touchscreen driver can potentially, but is not guaranteed to, produce forces the operating system to focus on a subset of possible touchscreen states to identify user interactions. Some touchscreens contain hardware support for the multitouch protocol while others depend on the driver to track the activities of different fingers. The way the operating system interprets these events can change over time which impacts cross-platform portability.

Touchscreen driver activity varies across Android devices. We performed a one finger swipe interaction across 26 physical Android devices. While performing the interaction on each phone, we collect touchscreen driver events through the Android `getevent` utility. We extract the first event packet corresponding to finger's initial contact with the screen and present the results in Fig. 4. Each row corresponds to a phone's touchscreen event packet and each column corresponds to a Linux touchscreen event. A dark blue tile indicates that that column's event was present in the device corresponding to that row. For example, the second device's event packet contains the `BTN_TOUCH`, `ABS_MT_TOUCH_MINOR`, `ABS_MT_POSITION_X`, `ABS_MT_POSITION_Y`, `ABS_MT_TRACKING_ID`, and `SYN_REPORT` events. The diversity amongst event packet contents limits portability across different devices.

Application Framework Version — Key decisions about Android's software architecture can change throughout operating system versions. These continuously evolving changes can even restrict recording touchscreen user interactions on one system and later replaying them on a similar device with a different Android version. For example, some systems are sensitive to whether the `BTN_TOUCH` event is present. Looking at the application framework source code between different Android versions, we find that different versions can expect different events from the touchscreen driver. Others have also reported similar experiences [3]. Additionally, vendors have access to the Android source code, which allows them to make modifications specific to the components they use.

Virtual Soft Keys — Many Android mobile device manufacturers choose to virtualize standard Android buttons on the device display instead of providing a physical button. For example, Fig. 3a shows that the MDP tablet uses virtual keys while the Samsung Galaxy S5 has hardware keys that are separate from the display. Virtualized keys use screen real estate to provide soft menus, which would have otherwise been available for the application to use. It implies that some of the quantization points for the touchscreen map correspond to the virtual keys, and as such record and replay needs to take this into account to ensure cross-platform portability.

III. Mosaic Design

We present the design and implementation of Mosaic, a cross-platform user input record and replay tool for Android. To overcome the fragmentation issues discussed in the previous section, Mosaic provides portability through virtualization. User inputs are captures on a host device which is then virtualized into a platform-agnostic intermediate representation which can then be retargeted for specific mobile devices.

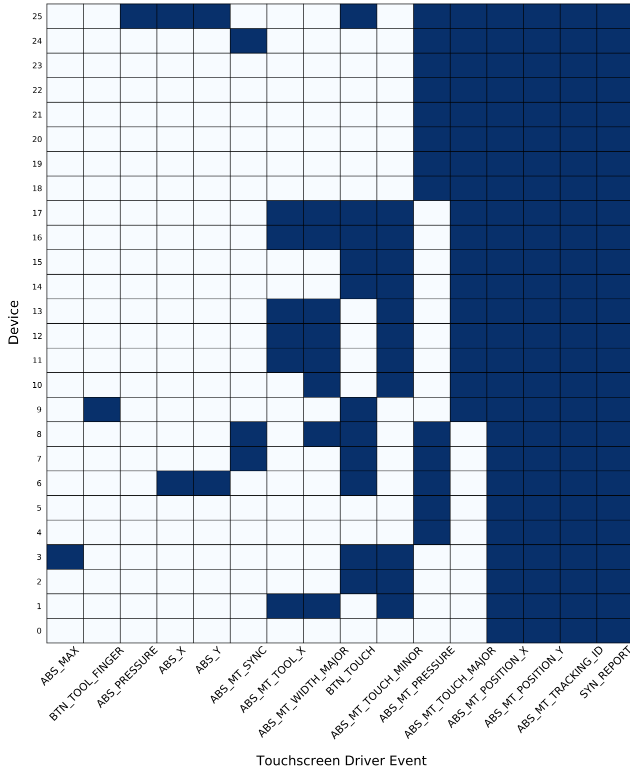


Fig. 4: Measured touchscreen driver fragmentation for finger press.

A. Formalizing Touchscreen Interactions

Mosaic abstracts touchscreen-specific input events into a set of user interactions on a virtualized touchscreen. To perform this conversion systematically we develop a formalization for touchscreen input events. Interactive mobile applications rely on user input (i.e. interactions) to drive program behavior. More formally, we denote this set of interactions as $\{I_1, I_2, ..., I_N\}$, where an arbitrary I_j corresponds to an individual interaction primitive – to be defined later.

While the touchscreen driver propagates user input information as individual events, we make the observation that the touchscreen implicitly packetizes these events into user interaction primitives. The Linux `/dev/input/event/` interface terminates a sequence of events with a `SYN_REPORT` event delimits different user inputs. Therefore, we define any arbitrary interaction primitive I_j as a set of events $\{e_1, e_2, ..., e_N\}$, where E_N is the only `SYN_REPORT` event in the set. Therefore, Mosaic relies on `SYN_REPORT` to identify interaction primitives throughout the sequential touchscreen event stream.

We identify three user interaction primitives that serve as Mosaic's platform-agnostic user interaction intermediate representation. The primitives consist of a finger press, release or movement. A finger initiates an interactions sequence when it first makes begins to *press* the touchscreen. Conversely, once the finger ceases to contact the touchscreen a *release* has occurred, terminating the finger's interaction sequence. In between the press and release, the finger may *move* across the touchscreen depending on the behavior (e.g. a swipe or pinch).

Our selection rationale for the primitives can be explained by the event packet diversity within Fig. 4. Across all the devices, a few key events are consistently hot: `ABS_MT_POSITION_X`, `ABS_MT_POSITION_Y`, `ABS_MT_TRACKING_ID`, and `SYN_REPORT`. Intuitively, these four events suggest that there is some minimal amount of state that is encapsulated within a finger press. The `ABS_MT_TRACKING_ID` event indicates a new finger interaction session and `SYN_REPORT` delimits each finger interaction packet within that session. Finger presses happen at specific touchscreen locations which are tracked using `ABS_MT_POSITION_X` and `ABS_MT_POSITION_Y`. We make similar observations for other user interactions and thus deem that several user interactions can be succinctly and portably captured in the form of a few interaction primitives.

B. Calibration and Training

Different interactions consist of different touchscreen events on different platforms. In order to classify a particular device's touchscreen interaction set, $\{I_1, I_2, ..., I_N\}$, into the platform independent primitives (i.e., presses, releases and moves), Mosaic requires a device calibration phase. Mosaic prompts the user to execute a sequence of known, well-defined user interaction sequence to extract relevant device parameters required for virtualization and translation.

Mosaic prompts the user to perform a known user interaction in a controlled manner to identify events that are unique to presses and releases. In our implementation, the user performs a single finger swipe with the device in its default orientation (e.g. portrait for smartphones and landscape for large tablets). This results in a set of user interactions, $\{I_1, I_2, ..., I_N\}$ where:

$$Press = I_1 \tag{1}$$

$$Release = I_N \tag{2}$$

$$Move = \cup_{j=2}^{N-1} I_j \tag{3}$$

which are used to isolate:

$$UniquePress = Press - Release - Move \tag{4}$$

$$UniqueRelease = Release - Press - Move \tag{5}$$

In addition, Mosaic uses the Android `dumpsys` utility to probe the device for its default orientation, display resolution and touchscreen quantization information, which will be used throughout the remainder of the record and replay framework.

We demonstrate the calibration phases as part of the end-to-end translation example in Fig. 5, shown in the "Calibration" column. Before recording the interactive trace for a slingshot pull within the Angry Birds gaming application, the source and target devices are calibrated. The Samsung Galaxy S5 (source) and Asus Nexus N7 (target) have different raw event streams that express the same interaction primitives.

The single-swipe calibration for the Samsung Galaxy S5 (source) results in 17 touchscreen events belonging to five distinct interactions, separated by each `SYN_REPORT`. The first seven events correspond to the $Press$ and the last three events correspond to $Release$, leaving the middle seven events as $Move$. Applying equations (4) and (5) produces $UniquePress$ = {`ABS_MT_TRACKING_ID BEGIN`, `BTN_TOUCH DOWN`} and $UniqueRelease$ = {`ABS_MT_TRACKING_ID_END`, `BTN_TOUCH UP`}, which are propagated to the virtualization stage. The same processes is performed for the Asus Nexus N7 (target) for translation.

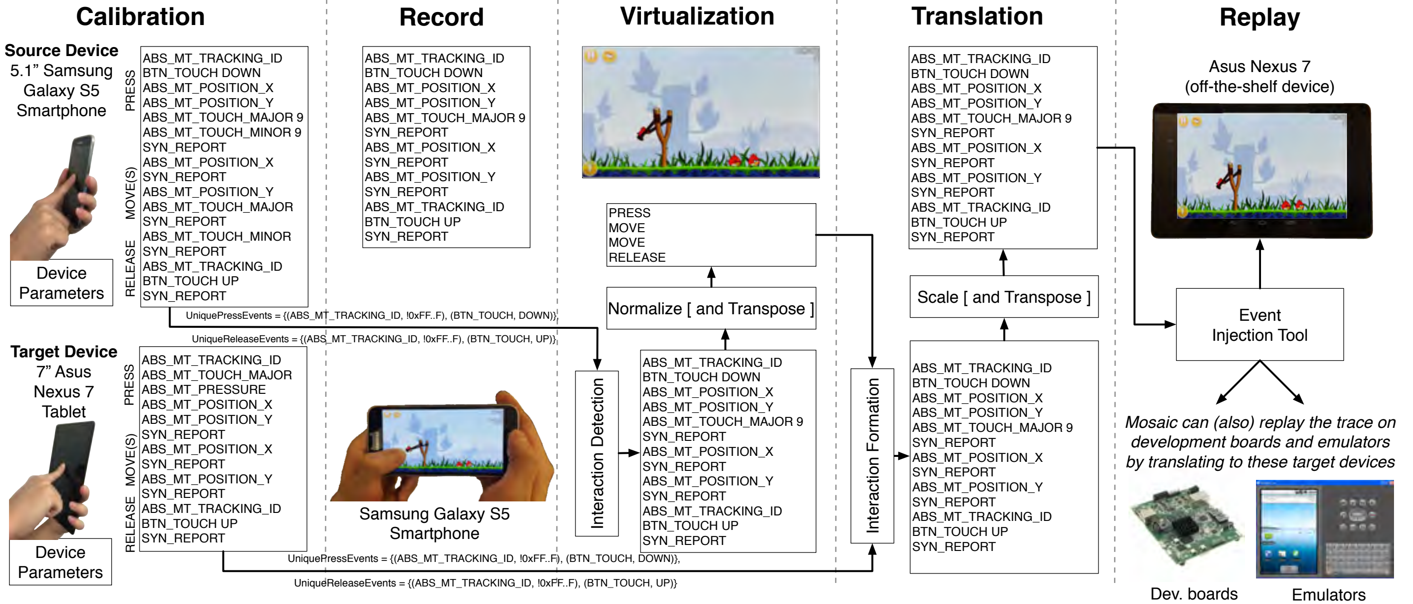


Fig. 5: An example showing the different phases of Mosaic's execution. Calibration allows Mosaic to extract key device specific parameters to enable cross-platform portability. It is an application independent phase that needs to be done only once per device under test. From thereon, users can record any application interactivity. Recording is transparent to the user. The virtualization and translation phases are behind the scenes' Mosaic phases that transform the platform specific trace into a platform independent trace that be replayed on any other device.

C. Recording

Mosaic records a user input event stream by filtering the output of the Android `getevent` utility. `getevent` dumps a human-readable form of the event structs being streamed to Linux `/dev/input/event/` interface. Each event is a pentuple consisting of `(timestamp, input device, type, code, value)`. Mosaic ignores the input devices that do not correspond to the touchscreen. The result is a file that contains a sequential stream of touchscreen input events `timestamp type code value` appended to device metadata related to the state of the mobile device throughout the recording. The "Record" column in Fig. 5 shows a simplified version of the touchscreen user input trace for the Angry Birds slingshot pull on the `S5`.

D. Virtualization

To provide portability amongst various target devices, Mosaic converts the user touchscreen input events from the source device into primitive interactions (i.e., press, move or release) on a virtual display. The primitive interactions are stored as a platform-agnostic representation that also can be used as a domain-specific language for scripting user interactions, similar to `Orangutan` [4] with the following syntax:

```
time finger <N> <interaction> [[x] [y]]
```

Where `N` is a particular finger (Android supports up to five) that can interact with the screen through one of the three primitive user interactions. The `x` and `y` arguments are expressed as normalized distances (percentages) from the origin (e.g., top-left corner) of the virtual display. Mosaic classifies a recorded interaction I_j as follows (an α of $\frac{3}{4}$ worked well for us):

- Press: if $|UniquePress \cap I_j| > \alpha |UniquePress|$
- Release: if $|UniqueRelease \cap I_j| > \alpha |UniqueRelease|$
- Move: otherwise

Interactions parameters (i.e., coordinates) are mapped to a virtual screen that shares the same orientation as when the application usage scenario was recorded. Depending on whether or not the application was used in its default orientation, the coordinate mapping from the source device may have to be transposed to align the target device's origin with the virtual screen's orientation. In addition, the interaction coordinates are normalized to their maximum values.

We demonstrate how the `Angry Birds` usage scenario is virtualized within the "Virtualization" column of Fig. 5. The `UniquePress` and `UniqueRelease` identified during the "Calibration" phase are used to identify the four interactions within the recorded `S5` touchscreen trace. Similarly, the device parameters found during the calibration and record phase are used to identify if the device had been rotated and compensate the interaction parameters accordingly. In this case, the Samsung Galaxy S5 has a portrait default orientation whereas Angry Birds is meant to be used in a landscape oriented devices. Mosaic detects that the device was rotated in compliance with Angry Birds user interface requirements.

As a result, the two move interactions correspond to different axis in the recorded trace than in the virtualized user interaction trace. The first move was recorded as a movement along the y-axis since the device was rotated, but was virtualized to be a move across the x-axis on the virtualized landscape display. Any interactions that involve coordinates are normalized to their maximum values to be expressed as a percentage of the virtual display.

E. Translate

Translation is the inverse of the virtualization phase. It maps the virtualized interaction trace into a stream of touchscreen events for the target device. Mosaic interprets each virtualized interaction into a packet of touchscreen events that correspond to the events that were observed in the target's calibration phase. Mosaic detects whether or not the target

TABLE II: Evaluation devices that exhibit fragmentation from one another.

Device (Key)	Form Factor	Screen Size	Default Orientation	App Resolution	Total Resolution	Quantization	Soft Menu	OS Version
Samsung Galaxy S3 (SGS3)	Smartphone	4.8”	Portrait	720x1280	720x1280	720x1280	No	4.1.2
Samsung Galaxy S4 (SGS4)	Smartphone	5”	Portrait	1080x1920	1080x1920	1079x1919	No	4.2.2
Samsung Galaxy S5 (SGS5)	Phablet	5.1”	Portrait	1080x1920	1080x1920	1079x1919	No	4.4
Asus Nexus 7 (AN7)	Vertical Tablet	7”	Portrait	1920x1104	1200x1920	1343x2239	Yes	4.4
Qualcomm MDP (QMDP)	Horizontal Tablet	10”	Landscape	1920x1032	1920x1080	1920x1080	Yes	4.2.2
SVTronics OMAP5 EVM (SO5-DEV) [5]	Dev. Board	24”	Landscape	1280x672	1280x720	1280x720	Yes	4.2.2
Olimex A20-OLinuXino-MICRO (A20-DEV) [6]	Dev. Board	24”	Landscape	1280x564	1280x720	1280x564	Yes	4.2.2
Asus Nexus 7 – Emulator (AN7-EMU)	Emulator	7”	Portrait	1920x1104	1200x1920	1343x2239	Yes	4.4
Google Nexus 10 – Emulator (N10-EMU)	Emulator	10.55”	Landscape	2560x1600	2560x1600	2560x1504	No	4.4.2

device screen will have to be rotated, transposing the interaction coordinates if necessary. The normalized coordinate values from the virtualized interaction trace are then scaled by the maximum value for each event on the target device.

The “Translation” column of Fig. 5 shows the steps that are involved for translating the virtualized interaction trace to the Asus Nexus N7. The parameters taken from the device during the calibration phase are used to create templates for the set of touchscreen events that correspond to each interaction. Each press roughly maps to the $Press$ events and to $Release$ events for a release. A move event maps to updating an x and/or y coordinate with a SYN_REPORT appended to it.

F. Replay

Mosaic relies on a client-side component to inject user interactions into the target device. In its current form, Mosaic utilizes an in-house, optimized version of the replay unit from the RERAN [18] framework, which is essentially a write queue for /dev/input/event/ that preserves timing by sleeping between writes. Mosaic generates the touchscreen event input sequence for the target platform and passes it to the RERAN replay utility, which writes the events to the touchscreen’s /dev/input/event/ file.

G. Limitations

As with any tool, Mosaic has limitations. Mosaic’s guaranteed functionality relies on three basic assumptions about the applications and platforms on which it can record and replay.

First, an application usage scenario can only be replayed accurately if the application itself delivers repeatable behavior. Application UIs that are either time-variant or random may require different user inputs the original recording. For example, recording a user completing the daily New York Times crossword puzzle has a lifespan of one day, since there will be a different puzzle displayed the following day. Many other puzzle-based games such as Sudoku or multi-player first-person shooter games exhibit randomness as well.

Second, Mosaic maps interaction coordinates between platforms using a series of transposes, scalings and normalizations, which relies on the target UI elements positioning to scale linearly with screen dimensions in a given orientation. While many applications abide by this, some application rescale, rearrange and even omit UI elements based on the screen resolution. For example, some manufacturers deviate from the default Android touchscreen keyboard for their devices to add additional keys. Since Android allows users to download customized keyboards, a solution would be to download the same scalable keyboard across the devices under study.

Third, in order for an application to be replayed correctly across platforms, the target platform’s performance must be fast enough to make the UI components available in time for the replayed user interactions. As an application is interacted with, UI elements may appear or views may change, and these must be completed in time to exercise the same functionality across different mobile devices. In general, with the exception of old smartphone devices that are extremely slow, Mosaic is able to provide robust cross-platform portability on this issue.

IV. Evaluation

Mosaic provides cross-platform portability for user interactions across a range of devices, operating system environments, user record and replay inputs. Mosaic works across a range of different device that have different screen sizes, resolutions, features etc. We test Mosaic on eight different platforms, including off-the-shelf smartphones and tablets, development boards and emulators. We tested over 50 applications found on the Google Play application marketplace, using different users and use-case scenarios. Under the previously stated application assumptions, Mosaic provides robust replay functionalities. As with any tool, it has shortcomings that could be overcame with more effort. Nonetheless, Mosaic and its source code will be made freely available online to be used and extended as developers and researchers see fit.

A. Experimental Setup

To effectively evaluate Mosaic’s cross-platform record and replay capabilities, we selected a diverse set of Android mobile devices that reflect the fragmented Android ecosystem. These devices are shown in Table II. The selected devices exhibit significant hardware- and software-diversity from another. The device screen sizes range from 4.8” to 10.55”, falling into all four mainstream mobile device form factor categories: smartphones, phablets and vertical and horizontal tablets. Similarly, screen resolution and touchscreen quantization are also broad.

Our selected mobile devices also exhibit different software characteristics that can impact record and replay. The latest three Android operating systems, at the time of writing, are present across our device set. The tablets also posses soft, virtual keys, such as menu and back buttons, that occupy a portion of the screen and touchscreen.

We also consider emulators and development boards for our evaluation because systems have emerged that incorporate

TABLE III: Mosaic on Angry Birds running on fully featured off-the-shelf mobile devices.

(a) Mosaic features fully enabled.

		Replay				
		SGS3	SGS4	SGS5	AN7	QMDP
Record	SGS3	✓	✓	✓	✓	✓
	SGS4	✓	✓	✓	✓	✓
	SGS5	✓	✓	✓	✓	✓
	AN7	✓	✓	✓	✓	✓
	QMDP	✓	✓	✓	✓	✓

(b) Mosaic without coordinate remapping.

		Replay				
		SGS3	SGS4	SGS5	AN7	QMDP
Record	SGS3	✓	×	×	×	×
	SGS4	×	✓	✓	×	×
	SGS5	×	✓	✓	×	×
	AN7	×	×	×	✓	×
	QMDP	×	×	×	×	✓

(c) Mosaic without touchscreen event remapping.

		Replay				
		SGS3	SGS4	SGS5	AN7	QMDP
Record	SGS3	✓	×	×	✓	×
	SGS4	✓	✓	×	✓	✓
	SGS5	✓	✓	✓	✓	✓
	AN7	✓	×	×	✓	×
	QMDP	✓	✓	✓	✓	✓

architectural simulators into the mainstream Android Open Source Project (AOSP) simulator [17]. Similarly, development boards are increasingly being used in computer architecture research due to the ease of instrumentation, and also because more information is publicly available about them. We use the emulation software to represent the Asus Nexus 7 tablet and Google Nexus 10 tablet. We also evaluate these devices in their native hardware form.

B. Coordinate and Interaction Mapping Sensitivity Analysis

We use the popular Angry Birds application and use it to demonstrate the important aspects about Mosaic. In particular, to reinforce our claim that fragmentation between both touchscreen coordinates and touchscreen event packets affects record and replay, we conduct a sensitivity analysis on the application. Table IIIa shows the results for recording and replaying Angry Birds across all our smartphone devices. The first row denotes the device on which the interaction was originally recorded on and the first column indicates the device on which the trace was replayed on with Mosaic. A (green) cell with a "✓" indicates a successful replay and a (red) cell with a "×" is a failure. All interaction recordings and replays across the entire device set are successful. We now reduce the Mosaic functionality to see how cross-platform record and replay is affected with coordinate and event remapping.

Touchscreen Coordinate Remapping — Remapping user interaction points across devices is important for Mosaic's ability to perform cross-platform record and replay. Table IIIb, arranged in a similar fashion as Table IIIa, shows how successful Mosaic is without mapping coordinates between the same devices; retaining the original device's unvirtualized coordinates instead. The diagonal "Y" shows that devices only replay an application with themselves without coordinate remapping. The exception to this is `SGS4` and `SGS5` because they share the same resolution and touchscreen quantization.

Touchscreen Events Remapping — Touchscreen driver differences between devices also affect cross-platform record and replay. Table IIIc shows the results of using Mosaic without remapping the target device's touchscreen event packets discovered in the calibration phase. The `SGS5` and `QMDP` touchscreen packets provide the most information which other devices can salvage. Contrapositively, the `SGS3` and `AN7` do not require a lot of touchscreen state to identify a user interaction, allowing many device inputs to map to them.

C. Application Record and Replay

Mosaic allows many popular Android applications to be recorded and replayed across a variety of mobile devices. We downloaded over 50 free applications from Google Play and report the applications we are able to record and replay across two of the mobile devices in Table IV. The applications include a wide distribution of applications such as games, social networking, news, readers, image editing and utilities.

We are able to replay 45 of the downloaded applications. Applications that were not replayable across platforms were mainly due to the limitations/assumptions outlined in Sec. III. For example, applications such as Candy Crush and Fruit Ninja, generate dynamic content that prevents the same user inputs from driving the same application behavior. We explored several cross-platform combinations of record and replay using real users. For instance, we took traces from `SGS3` and ran it on the `AN7`, and vice versa.

To avoid user-specific bias, we had different users exercise different applications. For instance, two individuals might have different finger swipes, touches, etc. Thus, we tested Mosaic against user-specific gesture behavior. We had five users use the applications to exercise Mosaic with different human behavioral patterns, which leads to different event packet types.

We also took traces on development boards and emulators and ran them on real hardware, and vice versa. However, we faced some additional technical obstacles – some of which we resolved and some which are still open problems. The `SO5` and `A20` development boards do not have touchscreens so we had to extend Mosaic to add support for the mouse input protocol. We were able to successfully record traces on the `AN7-EMU` and `N10-EMU` emulators and replay them on mobile devices, but were not able always able to do the converse. The emulator system clocks use the host system's clock instead of emulating system time which affects the playback accuracy. For similar reasons, the emulator is not able to process drags interactions. We feel that this problem can be solved either by modifying the virtual interaction trace to align with events in the emulator or for the emulator expose a separate system clock that can be used for injecting user interactions.

D. Replay Overhead Analysis

There are two sources of potential overhead. The first is during recording. We did not observe any "jank" [7] in user interactions during our recording. The second source of overhead stems from replaying the recorded trace. We measure the replay overhead for ten of the applications shown in Table IV, denoted with an asterisk. The overhead is always less than 0.2% except for `Flashlight`, which is 0.645%. The overhead is more because of the cold start penalties. Flashlight runs for a very short time, and therefore the program startup overhead cannot be sufficiently amortized. Nevertheless, it is negligible and does not affect user experience during replay.

There is a small amount of replay memory overhead. The replay mechanism relies on a pre-existing event injection tool (`replay.c` from `RERAN`), which pre-allocates memory for

TABLE IV: List of 45 popular applications from Google Play that we were able to successful replay across devices.

Adobe Reader*	Angry Birds*	Big Fish Casino	Bingo Fever - World Trip	Bingo - Secret Cities
Bubble Bee	Bubble Blaze	Bubble Witch Saga 2	Chase	Chrome*
Clash of Lords 2	Clean Master	Coin Dozer	Destiny	DH 2014
Diamond Digger Saga	Dolphin Browser	Five Nights at Freddys	Flashlight*	Flipagram
Google Play Newsstand	Hill Climb Racing*	iHeartRadio	Instagram	Jump Ninja
Legend of Master Online	Messenger	Minion Rush	MonsterWarlord	News & Weather
Firefox	Photo Grid	Piano Tiles	Slotomania	Slots Fever
Slots - House Of Fun	Solitaire	Spider-Man	Steel Avengers	Temple Run 2*
Tycoon Mania	Video Poker	Walmart	WatchESPN	YouTube*

all the events that need to be replayed in the translated file, just prior to execution. The amount of memory allocation varies from one run to another, since it depends upon the number of events in the translated file. In practice, however, we found that this does not affect performance in any noticeable manner. The average memory overhead is approximately 5.5 KB, which is negligible compared to the overall application code footprint. In total, the memory overhead is much less than one percent.

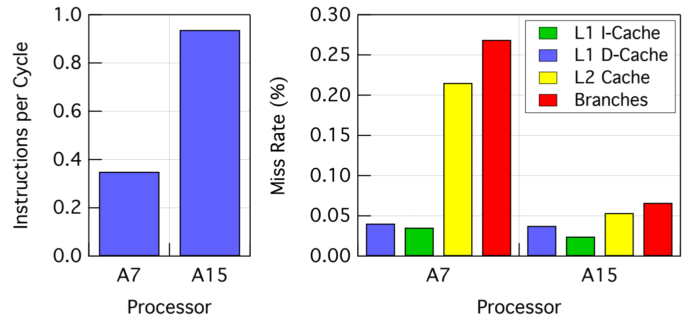
V. Example Use Case: Microarchitecture-level Performance Comparison

Recent mobile CPU design trends have begun to embrace heterogeneity. For example, ARM big.LITTLE technology pairs brawny "big" high-performance cores with weak "little" low-power cores to provide balanced multi-core CPU designs. In general, the big cores are meant to be used in situations where they significantly outperform little cores. Otherwise little cores are preferred to conserve battery life. Therefore it is important to know when an application can benefit from big cores because they waste energy compared to little cores.

In order to perform our analysis, we need to use Mosaic to collect a realistic user trace from a mobile device to study it with our experimental infrastructure. We recorded a user navigating to and playing the first level of Angry Birds on the `SGS5` that we replayed on the `SO5-DEV` and `A20-DEV` development boards. The `SGS5` allows us to capture a realistic user interaction trace which was not possible on the `SO5-DEV` and `A20-DEV` that use mice and monitors instead of touchscreens. We have to use the development boards for our study because the CPU hardware performance counters and superuser (root) access is disabled on the `SGS5`, which is standard practice for mobile device manufacturers. We recompile Android (4.4.2) for each development board to enable hardware performance counters to be monitored through the ARM Streamline [8] tool.

We did not have access to a single heterogeneous CPU to conduct our experiments on so we study each CPU separately using the methodology in [25]. Our experiment considers the `SO5-DEV`'s ARM Cortex-A15 CPU as the big cores and the `A20-DEV`'s ARM Cortex-A7 CPU as little cores. A15's and A7's are a common heterogeneous CPU pairing, including the `SGS4` and `SGS5` we use in our study. We use instructions-per-second (IPC) as our responsiveness metric because it portrays how efficient each microarchitecture is for executing the interactive episode. The A15 is a dual-core processor, each with a triple-issue, out-of-order superscalar 15-stage pipeline. The L1 I- and D-caches are each 64 KB and the L2 cache is 2MB. The A7 is also a dual-core processor, but each core consists of a dual-issue, in-order 8-stage pipeline. The L1 cache parameters are the same as the A15 but the L2 cache is only 512KB. The A7 and A15 are clocked at 1 GHz and 912 MHz, respectively, which are the closest frequency configurations available, on each.

Our results show that the A15 notably outperforms the A7 throughout the Angry Birds interaction trace. This suggests it is worth expending the extra energy to use the A15 over the A7 in order to maximize responsiveness. Although the A15 was configured to be clocked slightly higher (9.6 %) than the A7, Fig. 6a shows its IPC is 3X the A7's IPC. Fig. 6b provides a more in-depth look at why the A15 easily surpasses the A7's performance. The A15 has a 4X lower branch misprediction rate than the A7's branch predictor, allowing the program to feed the A15's triple-issue width and out-of-order pipeline. The A7 has slightly higher L1 I- and D-cache miss rates than the A15. The resultant pressure on A7's L2 cache along with its modest sizing compared to the A15, cause the A7's L2 cache to have a 5X higher miss rate than that of the A15 – making it a major microarchitectural bottleneck.



(a) Instructions per cycle. (b) Microarchitecture statistics.

Fig. 6: Using Mosaic to perform a microarchitecture-level performance analysis between ARM Cortex-A processors for Angry Birds under the same user interaction sequence.

VI. Prior Work

We discuss prior work related to record and replay, especially in regard to UI-level replay for Android devices.. We focus on UI testing and automation tools and how they interact with the Android input architecture in Fig. 1.

User-level UI Testing — Many developers and researchers test and study interactive mobile applications manually. Either they will test mobile application behavior or outsource the work to a third party. The advantage of this approach is that it is representative of real user behavior for the device under test since real humans are utilizing the application. However, humans react in terms of tens of milliseconds [19], while computer systems operate on the order of micro and

nanoseconds. For the same functional use case, there can be significant variation amongst users and across runs which can affect the soundness of experimental data.

Kernel-level Replay — Compared to other kernel-level application record and replay tools, Mosaic is the only one to able to be cross-platform. RERAN [18] is able to record and replay all many sensor inputs for a particular; however, the replay is tightly coupled to the phone it was derived from. It does not alter the input event stream to enable portability across platforms. Similarly, Orangutan [4] is a touchscreen event interpreter where the developer writes a script and passes it to a run-time to execute on a target platform. The parameters such as interaction coordinates must be known by the developer. However, Orangutan does attempt to adapt its inputs to targets by toggling between Linux touch and multi-touch protocols. While this extends it's reach, Orangutan is not able to detect and handle subtle event-level differences between all machines based on our experimental experience.

Application-level Replay — Google provides the uiautomator [9], monkeyrunner [10] and espresso [11] testing frameworks. uiautomator is a JUnit-based application testing framework. Monkey runner allows a developer to externally exercise an application. It is also capable of injecting events, but its coordinates must be known to the developer. Robotium [12] and Selendroid [13] are two other popular application testing frameworks derived from the Selenium web browser automation tool [14]. These frameworks hook into the application source code, which is not always available for several top applications. Mosaic does not require source code changes.

Computer System Record and Replay — In the traditional computer system and architecture research community, deterministic record and replay has been extensively studied in the context of shared memory multithreaded programs. The goal is to deterministically reproduce the same (or similar) thread or memory interleaving across different program executions to ensure program reliability [16], [21], security [20], and debuggability [22], [23]. The corresponding techniques often target background or batch processing applications and benchmarks, such as PARSEC [15] and SPLASH [24].

Mosaic has an altogether different target and goal. It targets mobile applications that are often highly interactive and user-facing and aims at reproducing the same user interactivity, which is essential to conduct mobile system research more effectively. Since Mosaic operates at the user input event level, it has much lower performance overhead ($<$0.1%) and storage overhead ($<$5.5 KB) as compared to system-level record and replay tools, which typically incur orders of magnitude performance and storage overhead due to checkpoint and logging.

VII. Conclusion

Mosaic is a cross-platform, timing accurate user interaction record and replay tool for Android. In a mobile devices ecosystem consisting of tens of thousands of unique device form factors, Mosaic's unique virtualization scheme abstracts away the hardware and software complexity related to user input to replay user interactions across a variety of different mobile devices. Mosaic can enable researchers and developers to perform cross-device performance evaluations and analyze different types of user interactivity. As the mobile device ecosystem becomes increasingly fragmented to meet the diverse needs of users, the need for tools like Mosaic and will become increasingly important.

Acknowledgment

We would also like to thank Jingwen Leng and Yazhou Zu for being Mosaic beta testers, as well as the anonymous reviewers for their helpful feedback. This work was supported by an Intel research grant. The opinions, findings and conclusions or recommendations expressed in this material are those of the authors and not necessarily reflect those of our sponsor.

References

[1] 18796 Different Android Devices According to OpenSignals Latest Fragmentation Report. [Online]. Available: http://thenextweb.com/mobile/2014/08/21/18796-different-android-devices-according-opensignals-latest-fragmentation-report/

[2] GSMARENA. [Online]. Available: http://http://www.gsmarena.com/

[3] Porting Touchscreen driver for Android. [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1328515

[4] Orangutan. [Online]. Available: https://github.com/wlach/orangutan

[5] OMAP5432 Processor-based EVM. [Online]. Available: http://www.ti.com/tool/omap5432-evm

[6] Olimex A20-OLinuXino-MICRO-4GB. [Online]. Available: https://www.olimex.com/Products/OLinuXino/A20/A20-OLinuXino-MICRO-4GB/open-source-hardware

[7] Web Latency Benchmark. [Online]. Available: http://google.github.io/latency-benchmark/

[8] ARM DS-5. [Online]. Available: http://ds.arm.com/ds-5/optimize/

[9] Uiautomator. [Online]. Available: http://developer.android.com/tools/help/uiautomator/index.html

[10] Monkeyrunner. [Online]. Available: http://developer.android.com/tools/help/monkeyrunner_concepts.html

[11] Espresso: a fun little Android UI test API. [Online]. Available: https://code.google.com/p/android-test-kit/wiki/Espresso

[12] Robotium. [Online]. Available: https://code.google.com/p/robotium/

[13] Selendroid. [Online]. Available: http://selendroid.io/

[14] Selenium. [Online]. Available: http://www.seleniumhq.org/

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT*, 2008.

[16] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," in *SOSP*, 1995.

[17] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Gemdroid: a framework to evaluate mobile platforms," in *SIGMETRICS*, 2014.

[18] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *ICSE*, 2013.

[19] J. A. Hoxmeier and C. Dicesare., "System response time and user satisfaction: An experimental study of browser-based applications," in *AMCIS*, 2000.

[20] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *ASPLOS*, 2008.

[21] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergiesa safe method to survive software failures," in *SOSP*, 2005.

[22] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou, "Flashback: A light-weight extension for rollback and deterministic replay for software debugging," in *USENIX ATC*, 2004.

[23] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: Diagnosing production run failures at the users site," in *SOSP*, 2007.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

[25] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *Proc. of HPCA*, 2013.