Flux: Multi-Surface Computing in Android

Alexander Van't Hof[†‡] alexvh@cs.columbia.edu
Hani Jamjoom[‡] jamjoom@us.ibm.com
Jason Nieh[†] nieh@cs.columbia.edu
Dan Williams[‡] djwillia@us.ibm.com

[†]Columbia University, New York, NY
[‡]IBM T.J. Watson Research Center, Yorktown Heights, NY

Abstract

With the continued proliferation of mobile devices, apps will increasingly become *multi-surface*, running seamlessly across multiple user devices (e.g., phone, tablet, etc.). Yet general systems support for multi-surface app is limited to (1) screencasting, which relies on a single master device's computing power and battery life or (2) cloud backing, which is unsuitable in the face of disconnected operation or untrusted cloud providers. We present an alternative approach: *Flux*, an Android-based system that enables *any* app to become multi-surface through app *migration*. Flux overcomes device heterogeneity and residual dependencies through two key mechanisms. *Selective Record/Adaptive Replay* records just those device-agnostic app calls that lead to the generation of app-specific device-dependent state in system services and replays them on the target. *Checkpoint/Restore in Android (CRIA)* transitions an app into a state in which device-specific information can be safely discarded before checkpointing and restoring the app. Our implementation of Flux can migrate many popular, unmodified Android apps—including those with extensive device interactions like 3D accelerated graphics—across heterogeneous devices and is fast enough for interactive use.

1. Introduction

Users increasingly own multiple mobile devices of various shapes and sizes, with a recent survey reporting an average of roughly three devices per person [59]. Accordingly, there is a trend to run applications (apps) on multiple devices or *surfaces*. For example, it is possible to begin a movie using the Netflix app on a phone and switch to a larger screen to continue watching. In general, we expect to see more *multi-surface* apps emerge, including (1) switching from a larger device to a smartphone to travel, (2) displaying from a mobile device to a projector, (3) switching to a different device when the battery is running low, or even (4) collaboratively using an app during meetings, allowing multiple people to view, modify, and contribute.

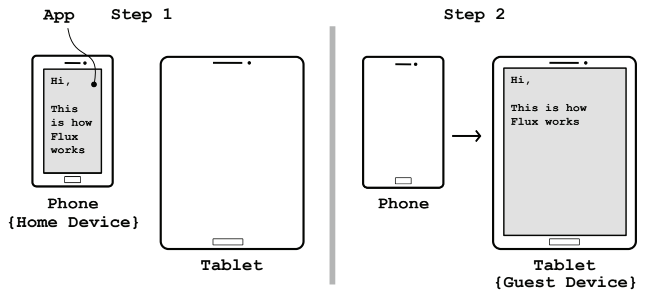

Figure 1. Multi-surface support through app migration: swipe to migrate unmodified app between paired devices without cloud support.

However, despite this growth, there is little system support for multi-surface apps. Today, there are two trending approaches. The first approach is *screencasting*, in which screen output from one device is sent to another [8, 9, 31, 37, 38, 44, 65, 66]. For example, Apple AirPlay [3] allows content on an iOS device to be displayed on an Apple TV. However, the app continues to run on the original device, still limited by its computing power and battery life. It cannot take advantage of the capabilities of the new device, such as CPU, GPU, or memory. Furthermore, apps often need to be explicitly written for systems such as AirPlay to achieve the best user experience. The second approach is to use a *cloud-based* approach in which the actual app content is stored in a back-end in the cloud. For example, iCloud or Google Drive and devices like Chromecast [23] make cloud back-ends pervasive. However, cloud-based approaches suffer from both a dependence on connectivity and a growing distrust of cloud providers to handle data. Cloud provider distrust is actually prohibitive in many enterprise environments with sensitive client data [5, 15].

We propose a third approach to achieve multi-surface computing: app migration. App migration enables the app to take advantage of the device in use, allows the original device to be used for other tasks, and does not require connectivity

EuroSys '15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741955

to a cloud provider; if disconnected from the Internet, devices can use ad-hoc networking. Furthermore, because it is implemented at the systems level, apps do not need to be written to support multi-surface operation.

Migration of an app is non-trivial. For example, in Android, even though apps are written expecting to be killed at any moment due to memory pressure, many apps do not automatically save all of their runtime state. If these apps crash, the state is lost. Therefore, it is not possible to migrate an app by simply killing it and starting it on the destination.

Furthermore, app migration between mobile devices is more complicated than many other environments due to device heterogeneity. Smartphones and tablets are tightly integrated hardware platforms that come in many different sizes and incorporate a plethora of devices using non-standard interfaces, such as GPUs and cameras. As of 2014, the OpenSignal database shows 18,796 different Android devices, up from 11,868 reported a year earlier [46].

Device heterogeneity complicates the usual challenges of *residual dependencies*, or state left in the source system after migration, in two ways. First, apps interact with *system services*, shared processes that may maintain app-specific and device-specific state. It is not feasible to migrate a shared system service along with the app or extract the app-specific state from the service. Even if the entire state of the system services was saved and restored on the target device, it may not work because the services manage device-specific state. Second, the running apps themselves contain—potentially device-dependent—state that is not easily accessible to the system. Blindly saving device-dependent app state and restoring it would not work across the thousands of different Android devices.

To address these problems, we introduce Flux, an Android framework for app migration. As shown in Figure 1, Flux enables any app to migrate directly from one device to another without any cloud support. Devices can be different smartphone and tablet hardware, and Flux ensures that once an application is migrated to a guest device, it is able to make full use of the guest device's hardware, including resizing and reformatting the application content to fit the display of the guest device. Flux accomplishes this seamless application migration across heterogeneous devices by introducing two novel mechanisms: *Selective Record/Adaptive Replay* and *Checkpoint/Restore In Android (CRIA)*.

Selective Record/Adaptive Replay eliminates residual dependencies due to system services. Specifically, during app execution, Flux interposes on app calls to system services and only records those that modify app-specific device state, automatically discarding stale interactions. Selective Record is also used to guarantee correctness of Android services after migration. During resume, the recorded app calls are *adaptively replayed* through Flux's service contextualization proxy to match the guest OS's system services. Importantly, this record/replay mechanism ensures that device-dependent state in the source is accurately recreated on the target.

CRIA checkpoints critical user- and OS-level state of the running app at the source and restores it at the target. A key feature of CRIA is that it integrates with Android to eliminate most residual dependencies on the system and customize the restoration of checkpointed state in a manner tailored to the target, supporting device heterogeneity. CRIA deals with device-specific state by putting the app into such a state that it discards much of the device-specific state on the source. Next, CRIA checkpoints core app state, including app-specific state in Android specific drivers such as *Binder*, the IPC mechanism through which apps interact with the system-provided services that front most devices, e.g., GPS and camera. On restore, Flux leverages Android app initialization mechanisms to inform the app of changes to hardware state so that app-specific device state can be reconstructed in a manner customized to the guest platform, including matching the UI to the screen size of the guest platform. Restoring checkpointed state reestablishes the app's Binder connections to system services, now at the target.

We have implemented and evaluated a working prototype of Flux on Android. Our results show that Flux successfully migrates a wide range of the top apps from the Google Play store across different smartphone and tablet hardware running different OS kernel versions. We show that the runtime overhead of Flux during app execution is negligible. Not surprisingly, the migration time is dominated by network transfer times. Nonetheless, we found that migration time and the amount of state transferred was modest in most cases, demonstrating that Flux is fast enough for interactive use.

This paper presents the design and implementation of Flux. Section 2 gives an overview of Android. Section 3 describes the Flux architecture, focusing on Selective Record/Adaptive Replay and CRIA. Section 4 presents experimental results. Section 5 discusses related work. Finally, we present some concluding remarks and directions for future work.

2. Android Background

Figure 2 gives an abridged overview of the Android system components that apps are dependent on and are therefore critical during app migration. An app in Android is written in Java and runs inside an isolated instance of the *Dalvik* VM. Typically, an app runs in a single process; less commonly, an app may be split into multiple processes. Apps are installed with the PackageManagerService, which tracks app metadata such as requested permissions. An app is typically isolated to a single data directory through filesystem permissions and access to storage such as an SD card requires explicit permission upon installation. An app consists of any number of *activities* and, when necessary, talks to *services* via Binder, Android's primary IPC mechanism. An activity is an app component providing a UI with which users can interact with to perform tasks, such as send an email, or dial the phone. A service is an app or system component that can perform long-running operations in the background without a UI. When migrating, the various device state associated with both activities and services must be correctly handled.

Android apps rely heavily on interactions with shared, long-running system services. For example, the NotificationManagerService allows apps to post notifications to the status bar, and the AlarmManagerService allows apps to schedule code to be run at some point in the future. Apps communicate with these services and with each other exclusively via Binder, either explicitly via RPC service interfaces or through *Intents*. Intents are messaging objects used to request an action from another app, which can be broadcast to all relevant apps by the ActivityManagerService. To simplify the creation of RPC service interfaces, the Android Interface Definition Language (AIDL) allows programmers to write an interface by simply defining method prototypes. AIDL will then generate the necessary serialization and IPC code required for the interface.

In addition to distributing Intents, the ActivityManagerService is responsible for managing the running of Android applications, including starting and stopping app components, and registering app-requested BroadcastReceivers, which act as listeners for apps for various events, e.g., informing them of WiFi status changes. Another duty of the ActivityManagerService is controlling the life cycle of activities. In Android, activities transition between various states of their life cycle. After creation, an activity enters the Resumed state, where it remains until it is sent to the background or another activity partially obscures it from view. Once sent to the background, the activity transitions to the Paused state. In this state, the activity no longer receives user input and cannot execute any code. If the activity is not quickly brought back to the foreground, the Android task idler will place it into the Stopped state. In this state, the activity is guaranteed to not be visible to the user and it will no longer be able to render its user interface.

The user interface of an Android app consists of a Window, provided by the WindowManagerService, for each activity. A Window, similar to a desktop window, contains a single Surface in which the content of the Window is rendered. This Surface will be destroyed when the app is in the Stopped state to conserve resources. Each Window also has a View hierarchy attached to it. View hierarchies are rooted by a ViewRoot and consist of ViewGroups containing Views, which are interactive UI elements. Each time a Window is to be rendered, the View hierarchy is traversed and each View draws its portion of the UI.

Communication with devices takes place via system-provided Binder services, e.g., the SensorService. An exception is the GPU, which is interacted with directly using the standardized OpenGL ES library that abstracts away hardware details. Similar to other hardware libraries in Android, OpenGL consists of both a generic library, presenting apps with a well-known API, and a vendor-specific library, implementing device-specific code called by the generic library.

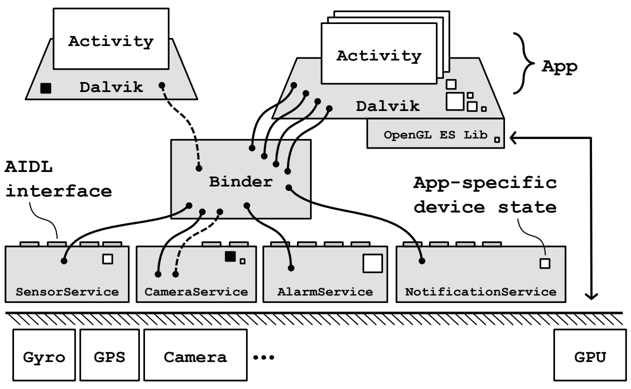

Figure 2. Overview of Android system components and their relation to one another through Binder.

All Android apps rely on the Android version of the Linux kernel: it is therefore a shared resource. In the kernel, Binder is implemented as a driver. Binder communication typically consists of clients talking to services. In Binder, the service side is dubbed a *node* and all clients reference nodes via process-specific *handles*, identified by a simple integer. Communication to another Binder node cannot occur without first being given a reference to it by the process who created it or a process already holding a reference to it. Therefore, services wishing to offer other processes an RPC interface must register themselves with the userspace *ServiceManager*. The ServiceManager maintains a registry of Binder references corresponding to names given when the service was registered. It is up to the service itself to decide whether or not a calling process has permission to make a particular RPC. Other features of the kernel include *ashmem*, a shared memory driver; *pmem*, a physically contiguous memory allocator used by devices like the GPU; an alarm driver, allowing the AlarmManagerService to schedule alarms that can trigger regardless of the machine's sleep state; *wakelocks*, a power management feature used to keep the machine awake while a wakelock is held and to sleep otherwise; and the Logger driver. When migrating between devices, the state of all these Android specific drivers must be considered.

3. Flux

We assume an environment that consists of many mobile devices running Flux. An app can be installed on some, but not necessarily all, mobile devices. The device on which the app is natively installed is called the *home* device. As shown in Figure 1, users in our environment can migrate any running app, along with all its active state, from its home

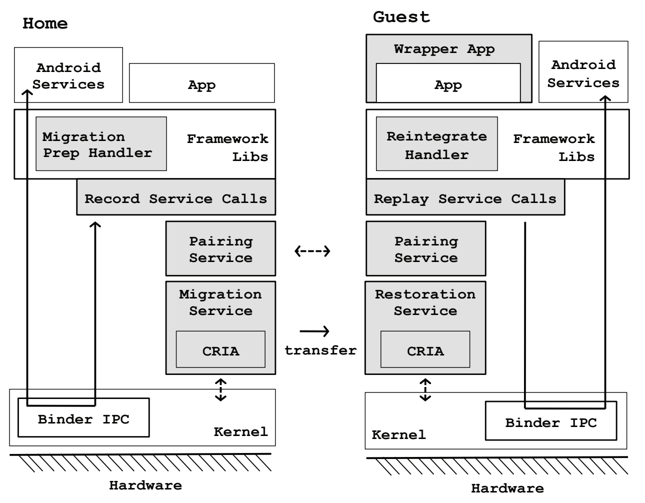

Figure 3. Overview of Android components involved in migration. Components added by Flux highlighted in gray.

device to other *guest* devices. We do not rely on any back-end (cloud) support or modifications to the app.

3.1 Migration Life Cycle

Figure 3 shows that Flux consists of a number of components, which are highlighted in gray. We describe the high level role of each component in the context of a migratable app's life cycle, as depicted in Figure 4: *Pairing*, *App Execution*, *Migration Out*, and *Migration In*.

Pairing. Before a user migrates an app from the home device to a guest device, Flux performs a one-time pairing operation that synchronizes the home device's core framework and libraries to a custom location on the guest's data partition. This is needed because the core framework and library binaries may differ across devices; the frameworks and libraries used by an app must remain the same before and after migration. The differences between these files are generally small. Consequently, the synchronization operation is performed efficiently since most files are linked against the identical files on the guest's system partition. In our current implementation, we use `rsync` for synchronizing files and its `--link-dest` option for linking identical files.

Similarly, Flux also verifies and synchronizes the home device's app binaries, known as Android Package Files (APKs), and app data files to the guest device. This includes any app-specific data directories residing on the SD card, but not general SD card data available to all apps with SD card access. Since apps may be updated frequently, the paired APK is verified prior to migration and updated if necessary.

As part of the pairing, Flux pseudo-installs the APK's metadata on the guest with its PackageManagerService. This allows the guest to be aware of the app's permissions and components but does not actually install the app data, such as the

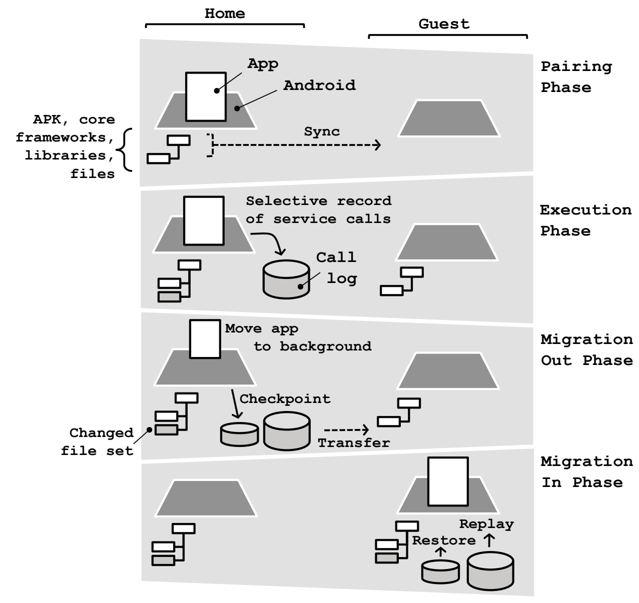

Figure 4. Workflow phases of Flux migration.

app executable and other resources. This pseudo-installed app acts as a wrapper when migrating in; additionally, it differentiates a migrated app on the guest device from the natively-installed version.

Due to the fragmentation of the Android market, app binaries are typically designed to run across a wide range of Android versions. However, if a particular APK requires an API level that is incompatible with the software stack of the guest device, Flux informs the user the app cannot be migrated.

App Execution. During app execution, Flux selectively records an app's interactions with system services through Binder's IPC mechanism. This recording functionality, described in Section 3.2, uses framework-level *decorators* of the system services' RPC interface. Additionally, the recording functionality is provided in core framework-supplied libraries and is transparent to the app. The recorded log is primarily used to restore the app-specific state of system services once the app has migrated to a guest device, avoiding the need to migrate these services along with the app. It is kept small by automatically discarding stale calls.

Migration Out. A user initiates a migration operation through a two-finger vertical swiping gesture. Flux's first step is to use Android's built-in mechanisms to free as much device-specific state as possible. Specifically, Flux instructs apps to go to the background, which helps free drawing surfaces. Then, Flux triggers a low-memory condition, which further releases graphic-related resources. Finally, Flux extends OpenGL to remove any remaining vendor-library-specific state.

Next, Flux checkpoints the app's process(es). Because the primary way in which Android apps interact with the rest of the system is through Android's Binder IPC mechanism, Binder IPC state must be saved as part of the checkpoint. Flux achieves this using CRIA, as described in Section 3.3. Flux's checkpoint includes not only per-process app state, but also the recorded log of calls made by the app to interact with system services. Once complete, the checkpoint image is compressed and sent to the guest device, along with the app's data directory.

Migration In. To restore the app from the checkpoint on the guest device, Flux uses the wrapper app created at pairing time as a shell in which to restore the checkpointed image of the migrated app. The wrapper app is launched in a private virtual namespace for process identifiers to ensure that app processes see the same identifiers even if the underlying operating system identifiers may have changed. The wrapper app is also jailed to the previously synchronized filesystem containing the home device's libraries and the app's APK. Once complete, Flux restores the app from the checkpointed image, as discussed in Section 3.3, including re-establishing the same Binder state for the app so that the app sees the same Binder handles. To complete the integration of the app into the new guest environment, Flux informs the app of any changes to hardware, and replays the recorded service calls, as discussed in Section 3.2, for the guest device's services to restore necessary state on behalf of the app, permitting the app to interact with system services right where it left off.

Flux does not restore the app's original network state, but rather leverages the fact that mobile apps are typically built around transient wireless connectivity and therefore are designed to correctly handle connectivity changes. When restoring a migrated app, Flux informs it of a loss of connectivity and availability of a new connection, allowing the app to handle the connectivity interrupt in the normal way. Finally, the app is brought to the foreground so that it becomes visible and available for use by the user.

3.2 Selective Record/Adaptive Replay

Because system services are shared by many apps and may have device-specific implementations, it is not desirable to migrate the system services themselves. Instead the corresponding system services on the guest device should take over after the app has been migrated. One approach could be to add checkpoint and restore hooks to every service an app may interact with, thereby enabling the extraction and restoration of a service's app-specific state. Doing so would be an overwhelming undertaking, requiring specialized knowledge of service and device implementation details that vary from one device to another. Instead of checkpoint-restore, Flux introduces *Selective Record/Adaptive Replay*.

In a straightforward implementation of record-replay to migrate app-specific state, all calls that update the app-specific state in system services are recorded, then deterministically

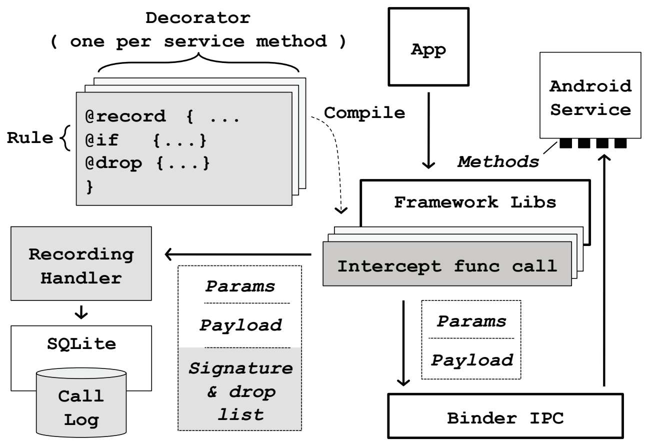

Figure 5. Selective Record and its place in the Android system. Flux-specific components highlighted in gray.

replayed on the guest device with its system services. This approach, however, has three key problems. First, system services support a wide range of features, each with different behavioral semantics. A one-size-fits-all approach to recording service calls will not suffice. For some services, particularly those that interact with the user, e.g., notification or alarm, simply recording and replaying all service calls would result in an incorrect state, e.g., the user would see past notifications that he has already acknowledged. Second, computing resources on a mobile device are scarce. A straightforward record-replay mechanism may unnecessarily record/replay all calls, wasting scarce mobile resources during app execution and migration and introducing an unacceptable latency waiting for the entire log to be replayed. Third, services across devices may not be identical and, in some cases, not available. A straightforward record-replay mechanism assumes a homogeneous environment and does not adapt to device variations.

To address these problems, Flux introduces Selective Record/Adaptive Replay to only record and replay calls to system services that are relevant to reproducing the current app-specific state in the respective services. As shown in Figure 5, since apps interact with system services via Android's Binder IPC mechanism, Selective Record simply interposes on the service interface calls used by Binder. These calls are based on standard APIs that are device independent, avoiding the need to understand device-dependent implementation details. Selective Record leverages the higher-level semantics available at these interfaces to identify which recorded calls are no longer relevant to the current app-specific state of a given service, and thereby can be discarded. Adaptive Replay then modifies the recorded calls in a way that best suits the characteristics of the guest device.

Selective Record. To capture the higher-level semantics from Android frameworks, Flux provides decorators that can be used by framework developers to instrument IPC ser-

SYNTAX	PURPOSE
`@record`	Indicate that calls to this method should be recorded.
`@drop` [method name], ...	Remove all previous calls to this method.
`@if` [arg], ... `@elif` [arg], ...	Qualifies `@drop` to only remove previous calls if all args given match.
`@replayproxy` [method]	When replaying, call proxy [method] instead of replaying the actual call.
`this`	A keyword representing the current method being decorated.

Table 1. Flux decoration syntax.

vice interface definitions. The decorators identify what calls should be recorded and how they affect the current state of the system. Our expectation is that the decorators are simple to use and require only minimal additions to existing frameworks. To further simplify the use of decorators, Flux takes advantage of interface definition languages (IDLs), which are commonly used to generate RPC interface serialization code. Flux extends the IDL to support decorators. Specifically, the Android IDL (AIDL), used for defining system service interfaces. For decorated interface methods, AIDL generates the necessary code to call our record function. The record function then asynchronously performs the actual recording and the necessary removal of stale calls which no longer affect the current state of services.

Table 1 lists the decorators that are supported by Flux. The syntax is modeled after Python's decorators, hence the name. Each decorator indicates what action should be taken with the subsequent call. There are four basic constructs. The `@record` statement indicates that calls to the respective function should be recorded to the log. The `@drop` statement indicates that previous calls to the respective function should be discarded from the log. The `@if` statement is used to qualify a `@drop` statement to only discard previously recorded calls from the log if all arguments provided with the `@if` statement match. Finally, the `@replayproxy` statement is used during replay to indicate that an alternative proxy method should be used instead of replaying the actual recorded call, thereby modifying the resulting replay.

Table 2 provides a full listing of all the decorated Android services along with the number of lines of code (LOC) required, separated into those that manage hardware devices and those that do not. For comparison purposes, it also shows the number of methods for each service interface, which provides a loose measure of the complexity of the respective interface. Generally speaking, services with larger interfaces require more lines of code to decorate. A few services are not yet decorated in the current Flux prototype, so their LOC are indicated as TBD. Most services require less than 50 LOC, except for ActivityService and AudioService, which require 130 and 150 LOC, respectively. As shown in Table 2, these two services also have larger interfaces than other services.

HARDWARE SERVICE	METHODS	LOC
AudioService	71	150
BluetoothService	202	TBD
CameraManagerService	8	31
ConnectivityManagerService	59	26
CountryDetectorService	3	5
InputMethodManagerService	29	37
InputManagerService	15	11
LocationManagerService	13	15
PowerManagerService	19	14
SensorService	6	94
SerialService	2	TBD
UsbService	19	TBD
VibratorService	4	26
WifiService	47	54
SOFTWARE SERVICE	**METHODS**	**LOC**
ActivityManagerService	178	130
AlarmManagerService	4	20
ClipboardService	7	6
KeyguardService	22	16
NotificationManagerService	14	34
NsdService	2	3
TextServicesManagerService	9	16
UiModeManagerService	5	9

Table 2. Decorated services in Android comparing the number of methods for each service interface and the number of lines of Flux decorator code for the service.

Example: NotificationManager. The NotificationManager, the AIDL interface for the NotificationManagerService, is used by apps to post and maintain notifications displayed on the status bar and in the notification drawer. It provides a simple example of how selective recording is performed. To migrate the app state, we must record these notifications to the guest device along with the app. Figure 6 shows a portion of an IDL defined interface derived from Android's actual NotificationManager, and Figure 7 shows the same definition with Flux decorations. The `@record` statement above `enqueueNotification` indicates that all calls to this function should be recorded. Inside the `@record` block above `cancelNotification`, the `@if` statement indicates that the n-tuple `(id,)` will be used as a signature to determine if a call to `cancelNotification` matches the signature of any previous calls to methods in the `@drop` list. The `@drop` statement contains a list of interface methods whose effect on the device state will no longer matter if `cancelNotification` is called with a matching signature. If a signature matches, any matching previous calls will be removed from the record. `this` is a keyword in the drop list, indicating that the call to the decorated method `cancelNotification`, should not be recorded if there is a match. Note that, because of the simplicity of this example, the decorations comprise a substantial portion of the resulting lines of code of the interface. However, this represents a small percentage of the total number of lines of code

```
interface INotificationManager {
    void enqueueNotification(int id,
             Notification notification);
    void cancelNotification(int id);
}
```

Figure 6. Simplified interface definition for Notification-Manager.

```
interface INotificationManager {
    @record
    void enqueueNotification(int id,
                   Notification notification);

    @record {
        @drop this, enqueueNotification;
        @if id;
    }
    void cancelNotification(int id);
}
```

Figure 7. Simplified interface definition for Notification-Manager with Flux decorations.

generated by AIDL to implement the interface, and it also represents an even smaller percentage of the total number of lines of code to implement the actual service.

Adaptive Replay. Once an app has been migrated to a new device, changes to hardware state that can normally be modified by the user, such as the WiFi state, are replayed to any listeners the app has set up. Should the guest device not contain hardware that was previously in use, e.g., GPS, the user is given the option to allow communication with that device to continue to take place over the network.

To alter the replay as needed, the `@replayproxy` statement may be used to decorate service methods to indicate that when a particular method is called during replay, an alternative proxy method should be used instead. For example, a proxy method could be used to adjust volume levels of music being played in accordance with the relative volume level differences between the home and guest devices. This approach is specifically used to support services like the AlarmManagerService.

Example: AlarmManager. The AlarmManager, the AIDL interface for the AlarmManagerService, is used by apps to schedule tasks to be run at some point in the future. It provides an example of how an alternative proxy method is used on replay. In this case, knowing only the arguments to methods is insufficient for deciding which calls must be replayed. This is because alarms are set through an API call, but then typically expire with time, not by being explicitly removed through a subsequent API call. To set an alarm, an app calls the AlarmManager's `set` API method, specifying a time for the alarm to go off and an Intent to be broadcast at that time.

```
interface IAlarmManager {
  void set(int type, long triggerAtTime,
           in PendingIntent operation);
  void remove(in PendingIntent operation);
}
```

Figure 8. Simplified interface definition for AlarmManager.

```
interface IAlarmManager {
  @record {
    @drop this;
    @if operation;
    @replayproxy \
        flux.recordreplay.Proxies.alarmMgrSet;
  }
  void set(int type, long triggerAtTime,
           in PendingIntent operation);

  @record {
    @drop this;
    @if operation;
  }
  void remove(in PendingIntent operation);
}
```

Figure 9. Simplified interface definition for AlarmManager with Flux decorations.

The app will have registered a BroadcastReceiver to listen for this Intent in order to accomplish whatever task the alarm was set for. If the app is not currently running when the alarm expires, it will be started prior to the Intent broadcast. To prematurely cancel an alarm, an app can call the `remove` API method, specifying the Intent previously passed to `set`. When migrating an app we must also migrate any previously set, and still active, alarms. Figure 8 shows a portion of an IDL defined interface derived from Android's actual AlarmManager, and Figure 9 shows the same definition with Flux decorations. The decorations indicate that calls with the same *operation* argument to `set` and `remove` should be dropped from the record as either the alarm has been removed or replaced with a new alarm and the previous calls are no longer necessary or valid. However, if an alarm is set and not removed but triggered by the advancement of time, it is important to detect that the alarm has already been triggered and should not be triggered again. To handle this common case, the `@replayproxy` statement is used to indicate that when replaying calls, our `alarmMgrSet` method should be called instead of simply replaying the call. This method, as shown in Figure 10, will first verify if the alarm is still active and, if it is, replay the call using Java Reflection. The method compares against the time of checkpoint rather than the current time to avoid missing an alarm set to trigger while the app was mid-migration. This ensures that an alarm that is set for after the time of checkpoint will be triggered as intended after migration.

```
void alarmMgrSet(Class alarmMgrClass,
                 Object newAlarmMgr,
                 String method, int type,
                 long triggerAtTime,
                 PendingIntent operation) {
    if (triggerAtTime <= checkpointTime)
        return;

    Method set = alarmMgrClass.getMethod("set");
    set.invoke(newAlarmMgr, type,
                 triggerAtTime, operation);
}
```

Figure 10. Simplified proxy method for replaying IAlarmManager.set().

Example: SensorService. The SensorService is used by apps to receive events from sensors, e.g. accelerometers, gyroscopes, etc. It provides another example of using alternative proxy methods on replay. In this case, API calls return handles to objects, such as Binder objects and socket descriptors, which are used by apps; these return values are uncommon in app-facing Android system services. To receive sensor events, an app asks the SensorService for a SensorEventConnection via its `createSensorEventConnection` method. The SensorEventConnection is a Binder object with an interface of its own that allows the app to enable desired sensors and receive a Unix domain socket via a call to `getSensorChannel`, over which it will receive the sensor events on via the SensorService.

When replaying calls to the SensorService, SensorEventConnection objects must be restored. This requires that the calls return the same handles to SensorEventConnection objects that the app was using before migration to ensure that the app continues to function properly after migration. Specifically, the Binder handle representing a SensorEventConnection and its respective Unix domain socket descriptor should remain the same after migration. To do this for the Binder object, a `@replayproxy` method is created for replaying the `createSensorEventConnection` call. The arguments supplied to this proxy method include the return value of the recorded call (the Binder handle representing a SensorEventConnection). This allows the proxy method to call the new device's SensorService to receive a new SensorEventConnection and map it to the correct Binder handle. Previously recorded calls to the SensorEventConnection will then be replayed. Similarly, to maintain the same descriptor for the Unix domain socket, a `@replayproxy` method is created for the SensorEventConnection's `getSensorChannel` call. This proxy will make the same call to the new SensorEventConnection's getSensorChannel method, obtaining a new connection with the SensorService (and by extension the Sensor). It will then `dup2` this descriptor into the original socket descriptor, reserved during restoration of the app.

Table 2 shows that although there are only 6 methods for the SensorService, it requires over 90 lines of code to decorate. The extra complexity here is due to the fact that this service is written natively in C++ and AIDL does not support generation of native code. The record/replay code that would normally be generated automatically through Flux's decoration syntax must be written by hand, requiring more care and time than would otherwise be needed. In the future, AIDL can be extended to support generating native C++ code [45].

3.3 Checkpoint/Restore In Android (CRIA)

To support migration of an app's processes, Flux extends traditional checkpoint-restart mechanisms [21, 25, 33, 47, 49] in a manner that leverages the characteristics of Android to save the core state of the app on one device and restore it on another; we call this Checkpoint-Restore In Android (CRIA). There are four types of app state to consider for checkpointing: process, device, filesystem, and network state. As discussed in Section 3.1, filesystem state is synced across devices and network state is simply re-established on the guest device after migration so that it appears simply as a loss of connectivity to apps, which are expected to handle such interruptions on mobile devices. We focus here on checkpointing process and device state.

Process State. CRIA builds on the Checkpoint/Restore in Userspace (CRIU) project [47], which is supported in the mainline Linux kernel. Hooks in the kernel allow CRIU to transparently obtain and inject all necessary internal kernel state required to represent the state of a running process. As part of restarting the app after migration, the app is encapsulated in a private virtual namespace [49] to ensure that operating system resource identifiers such as process identifiers remain the same, even if the same numerical identifiers are already in use on the guest system.

CRIA extends CRIU to take into consideration Android-specific device drivers: Binder, Logger, ashmem, pmem, and wakelocks. Of these, Binder required the most support. As shown in Figure 11, to capture dependencies that result from the use of Binder, CRIA checkpoints and restores three types of Binder connections: (1) internal app, (2) external system services, and (3) external non-system services. App processes contain handles that refer to various Binder connections. CRIA checkpoints the Binder state of each app process, including Binder handles, references and buffers, and notes which references are internal versus external to system services, including recording the association between references to system services and those service names.

The restore process is different depending on the type of connection. For Binder connections that are internal to the app, CRIA restores both ends of the connections. For Binder connections between the app and external system services, CRIA establishes new Binder connections with the same system services running on the guest device. CRIA asks the ServiceManager on the guest device for references to the equivalent new system services and injects those references in Binder with the previously issued handle identifier. For example, if the app references the NotificationManagerService using reference $id = 2$, it can continue to do so even on a new device with a different NotificationManagerService. This process only restores the connection between an app and various system services via Binder. As described in Section 3.2, app-specific state maintained by system services is restored via Selective Record/Adaptive Replay.

It is also possible that an app may have external Binder connections that connect to non-system services, such as non-system apps. A variety of solutions are possible to address this case, including migrating both connected apps or tethering the migrated app back to the home device. However, we have not encountered any such apps. For simplicity, CRIA currently checks for whether such Binder connections exist and if so, informs the user that the app cannot be migrated.

Support for the other Android-specific device drivers, Logger, ashmem, pmem, and wakelocks was relatively straightforward. Adding support for the Android Logger driver required few changes since the device is used like any regular file and does not persist per-process state. Although direct support for ashmem is straightforward to implement, its use is limited. ashmem is primarily used by Dalvik to name memory regions. For the sake of simplicitly, we modified Dalvik to use `mmap` for obtaining memory instead of ashmem. After this, we did not encounter other instances of apps using ashmem at the time of checkpoint, so direct support for ashem was not needed. Similar to ashmem, CRIA support for pmem is not necessary due to freeing resources prior to checkpointing. Finally, CRIA support is not needed for wakelocks and alarms as these are only used by Android system services; therefore, their process-specific state is handled by Selective Record/Adaptive Replay.

Device State. Checkpointing device-specific state is especially difficult on mobile devices because of the lack of hardware standards in these vertically integrated platforms. In Android, there are two cases: (1) devices are used indirectly by apps via system services that manage those devices, and (2) devices are used directly by apps. As described in Section 3.2, Selective Record/Adaptive Replay addresses the migration of device state in the first case.

For the second case, the GPU is the only device used directly by Android apps. Migration of graphical context is difficult given the complexity of the hardware and software and the substantial amount of app and device-specific state involved. However, because using the GPU involves consuming substantial system resources, most mobile operating systems have support for dynamically removing and restoring GPU-related resources. CRIA leverages and extends this support to avoid the need to checkpoint and restore GPU-

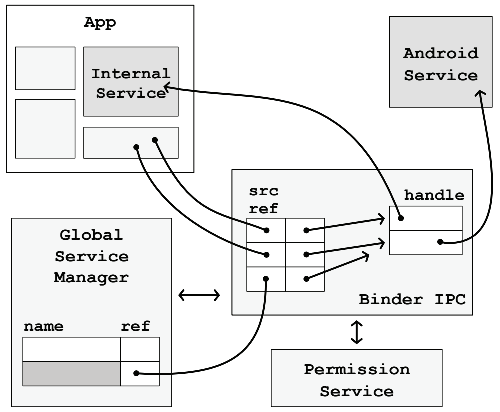

Figure 11. Binder dependencies captured with CRIA.

related state, dramatically simplifying the management of device state for migration. CRIA repurposes three types of Android mechanisms: background execution, low-memory condition, and conditional initialization.

CRIA leverages Android's background execution mechanism by instructing apps to revert to running in the background prior to being migrated. Because background apps are not visible to the user, various state associated with the visible interface of apps is not needed. By having an app run in the background, CRIA causes at least a partial removal of drawing surfaces and contexts corresponding to the visible state of an app. However, other graphical hardware resources and OpenGL contexts will still be retained.

To eliminate the dependencies on the GPU hardware, CRIA leverages Android's low-memory mechanisms, which can force apps to free graphics-related resources. CRIA invokes a trim memory request for the migrating app with the highest severity level via Android's ActivityThread's `handleTrimMemory` method. `handleTrimMemory` requests that the WindowManager trim its memory via a `startTrimMemory` RPC method. This invokes the HardwareRenderer's `startTrimMemory` method causing its caches to be flushed, and then invokes all ViewRoots' `terminateHardwareResources` method. This then calls the HardwareRenderer's `destroyHardware-Resources` and destroy methods causing all hardware rendering resources associated with those ViewRoots to be destroyed, the Canvas removed, and disables the renderer. ActivityThread will then call WindowManager's `endTrimMemory` method, which in-turn terminates all OpenGL contexts causing the HardwareRenderer to terminate and uninitialize OpenGL once all contexts are gone. The ViewRoot of the app is also destroyed, removing device-specific state that reference the ViewRoot.

Once completed, this leaves only a small amount of lingering native, graphics-related, vendor-library specific initialization state that must be removed. To do so, we extend Android's native OpenGL library with an `eglUnload` function. This is called after the HardwareRenderer is terminated and is used to completely unload the linked vendor-specific graphics libraries which are tied to the specific graphics hardware on the respective device, allowing for any new vendor-specific OpenGL library to be loaded when necessary.

Once an app is migrated and is being restored, CRIA leverages conditional initialization used by Android. Because Android is event-driven, various state used by apps is initialized on demand at time of use by checking first if the state is initialized before using it. CRIA reinitializes graphical context via the same initialization routines as used when starting an app. It takes advantage of conditional initialization to ensure that initialization is performed automatically due to the state of all objects appearing as if they were just created. Once graphics objects have been recreated and/or initialized, all Views will be in an invalid state, forcing them to be redrawn as they were prior to migrating. An important benefit of this approach is that, because graphics state is reinitialized and redrawn on the guest device, the resulting device-specific state is customized for the guest device.

3.4 Discussion

In our design, we made several decisions to simplify the system's role in managing consistency between devices. At the same time, we considered the impacts of the diversity of the Android framework and "future-proofing" Flux against changing versions.

Native vs. Guest Apps. Our current design differentiates native apps from migrated apps. This is because one cannot easily, and may not desire to, merge two running app instances, one that could be running natively and one that is being migrated. Thus, until the migrated app is brought back to its home device, an icon for the migrated app will exist on the guest device's launcher screen allowing for the user to resume the migrated app even after its been stopped.

Cross-Device App State Consistency. Once an app is migrated, it is guaranteed to have the latest and consistent snapshot of app state. When the user is finished with the app on the guest device, he may initiate a migration of the app back to its home device, thus resolving the inconsistency of app state between the two devices. If the user attempts to start the migrated app on the home device without having migrated it back, he is prompted with a message asking if he would like the app state from the guest device to be synced back to the home device or proceed while losing modified state on the guest device. Until an app has been migrated back to its home device, any security credentials allowing it to access online accounts will persist on the guest device until expiration or manual revocation.

Supporting Different Android Versions. Flux is capable of migrating apps between different kernel versions and minor Android version differences. Support for migration across major versions of Android would need to address two key challenges. The first is that an app using features only found in a newer Android API will be unable to migrate to an older version lacking those features. It would be difficult to surmount this obstacle and doing so would likely place a dependency on the source device, e.g., require that the target device continue to use some of the source's services over the network. The second is that the private APIs of services used internally by the framework must maintain backward compatibility with previous versions. Currently, these APIs are commonly changed by Google in between versions.

Limitations. Apps that request their OpenGL context persist while in the background are unsupported by Flux. Apps are able to do this in Android by calling GLSurfaceView's `setPreserveEGLContextOnPause` method. Doing so allows them to cache textures, shaders, etc. in graphics memory so there is no display delay once the app moves back into the foreground. The downside of this is that the app consumes resources even while not visible and as such the feature is not commonly used. Unfortunately, if the context never goes away and apps expect it to remain, they may not use conditional reinitialization relied upon by Flux. Completely unloading and reloading graphics state becomes problematic in this case.

Apps that request to be run in multiple processes are currently unsupported by Flux. Because multi-process apps are relatively rare, this feature was simply not yet implemented. It can be added with modest additional engineering effort, as CRIU already supports checkpointing an entire process tree.

Migrating an app while it is interacting with a ContentProvider is currently unsupported, e.g., when an app is receiving data after querying the system for contacts information. In Android, data intended for use by multiple apps, such as contacts, can be shared using ContentProviders. ContentProviders expose an API similar to databases, with methods such as query, insert, and delete. This API is accessible via Binder and ContentProviders are essentially Binder services with short-lived app connections. As such, it should be possible to leverage Flux's Selective Record/Adaptive Replay for support, but due to the limited time frame during which an app is typically interacting with ContentProviders and the likelihood of it interfering with migration, we have not yet implemented or exhaustively explored support for this.

Only app-specific SD card data directories are migrated along with an app. Due to this, apps accessing common SD card data at the time of migration will fail to migrate. Due to the potential size and quantity of files on the SD card, transferring them all is undesirable. Automatically transferring any open SD card files along with the app would allow these apps to migrate successfully, but any other common SD card files they were expecting to access would no longer be available. A potential solution could be to migrate the app and mount the home device's common SD card data as a network file system prior to restoring it, but this may not give the user the desired, or expected behavior.

Applying Flux to other mobile platforms. Although Flux is tailored to Android, the general design is applicable beyond it. Flux relies on three key platform characteristics: devices are utilized through system services and interacted with through a single IPC mechanism, app graphical resources can be released while the app is in the background, and the availability of an extensible checkpoint/restore mechanism. The first is a common mobile OS design paradigm. The third can always be overcome through engineering effort, and should be available for most Linux-based mobile OSes through CRIU. The second is perhaps the most problematic as any OS that does not already operate in this manner cannot easily be changed without breaking existing apps. For example, although iOS disallows apps from making OpenGL calls while in the background, apps are allowed to, and commonly do, retain their GL context. Removing their context while in the background would likely break most iOS apps. Existing work on checkpointing and restoring OpenGL state could be leveraged and improved upon to work around this requirement [30].

4. Evaluation

We have implemented a Flux prototype in Android and demonstrated its complete functionality in migrating unmodified Android apps across different Android devices, including the LG Electronics produced Google Nexus 4 phone and different hardware versions of the ASUS produced Google Nexus 7 tablet. The prototype has been tested to work with multiple versions of Android, including KitKat, the most recent version at the time of our evaluation. In migrating apps across devices with different screen sizes, Flux seamlessly migrates Android apps from home to guest device, including refreshing the app display to match the resolution of the target device.

We quantitatively measured the performance of our unoptimized prototype migrating and running a wide range of popular Android apps from Google Play. Our measurements were obtained using a Nexus 4 phone (Qualcomm Snapdragon S4 Pro APQ8064, Adreno 320 GPU, 2 GB RAM, 768x1280 pixel IPS LCD), a Nexus 7 (2012) tablet (NVIDIA Tegra 3 T30L, ULP GeForce GPU, 1 GB RAM, 1280x800 pixel IPS LCD), and two Nexus 7 (2013) tablets (Qualcomm Snapdragon S4 Pro APQ8064, Adreno 320 GPU, 2 GB RAM, 1920x1200 pixel IPS LCD). The Flux implementation used for our measurements was based on the Android Open Source Project (AOSP) version 4.4.2, the most recent version available at the time our measurements were taken.

To measure the cost of migration, we installed and ran eighteen different apps from the listing of top free Android apps from Google Play, including Candy Crush Saga, the longstanding most popular free game on Android. Figure 3 lists the apps we used, along with a brief description of the workload used for each app. To demonstrate the ability of Flux to migrate across heterogeneous Android devices, we migrated these apps across all four Android devices in four different combinations: (1) Nexus 7 (2013) tablet to Nexus 7 (2013) tablet to show migration using the same type of device on both sides, (2) Nexus 4 phone to Nexus 7 (2013) tablet to show migration from a smaller screen phone to a larger screen tablet, (3) Nexus 7 tablet to Nexus 7 (2013) tablet to show migration across two devices with very different hardware (GPUs, etc.) and kernel versions (3.1 and 3.4, respectively), and (4) Nexus 7 tablet to Nexus 4 phone to show migration from a larger screen tablet to a smaller screen phone, again with very different hardware and kernel versions. All devices were connected to a campus WiFi network. Before performing any migrations, all four devices were paired with one another.

NAME	WORKLOAD
Bible	View page of the Bible
Bubble Witch Saga	Play witch-themed puzzle game
Candy Crush Saga	Play candy-themed puzzle game
eBay	View online auction
Flappy Bird	Play obstacle game
Surpax Flashlight	Use LED flashlight
GroupOn	View discount offer
Instagram	Browse a friend's photos
Netflix	Browse available movies
Pinterest	Explore "pinned" items of interest
Snapchat	Take photo and compose text
Skype	View contact status
Twitter	View a user's Tweets
Vine	Browse a user's video feed
Subway Surfers	Play fast-paced obstacle game
Facebook	Post comment on news feed
WhatsApp	Send text to friend
ZEDGE	Browse ringtones and select one

Table 3. Top free Android apps and how they were used prior to migrating.

Before and after each migration, a user used each app on the respective device based on the respective app workload. All but two of the apps, Facebook and Subway Surfers, were migrated successfully across all four different device combinations, with the visual layout of each app adapted to the screen size of the respective device after migration. Facebook could not be migrated because it is one of the few apps that is multi-process, and the Flux prototype currently does not support multi-process apps. Subway Surfer could not be migrated because it requests that its EGL context persist, a limitation discussed in Section 3.3. We provide detailed measurements for migrating the other sixteen apps to quantify the cost of migration.

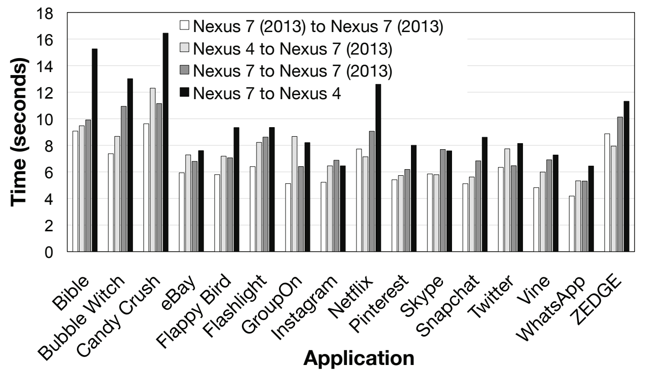

Figure 12. Overall migration times.

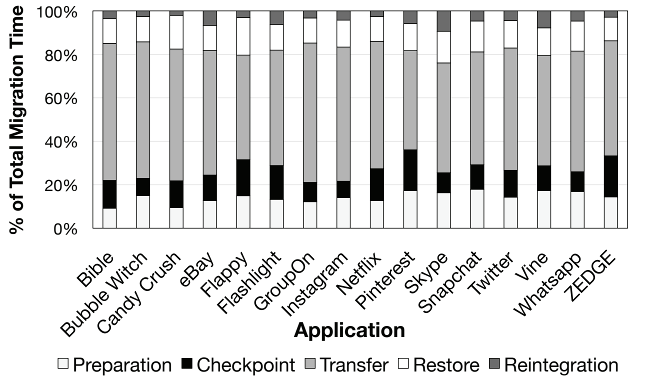

Figure 13. Breakdown of time spent during migration.

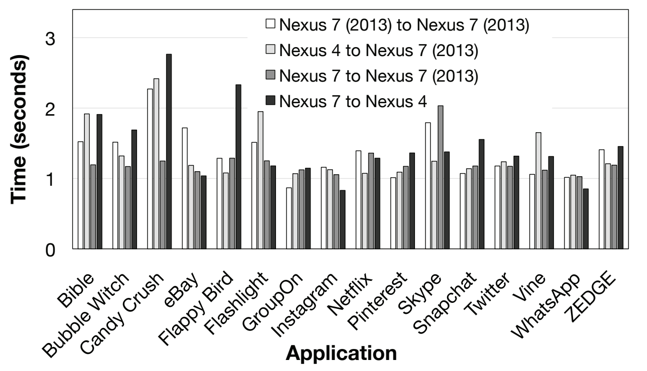

Figure 14. User-perceived migration time excluding data transfer phase.

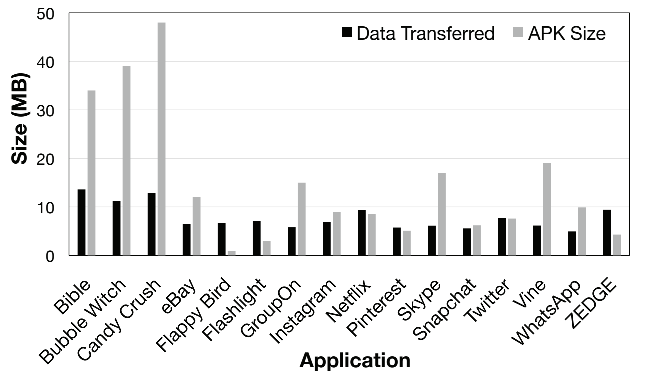

Figure 15. Amount of data transferred during migration.

Figure 12 shows the time required to migrate each of the apps across all four device combinations. Figure 13 shows the percentage breakdown of average migration times across the four device combinations. We breakdown the migration time into five stages: (1) preparation involves putting the app in the background to eliminate app-specific device state, (2) checkpoint involves checkpointing the app and its recorded log, (3) transfer involves verifying and syncing necessary file system state and sending the checkpoint image from one device to the other, (4) restore involves restoring the app from the checkpoint image, and (5) reintegration involves replaying calls to system services and bringing the app back to the foreground. The relative cost of each migration stage is fairly constant, with data transfer time dominating the cost of migration. As shown, over half the time on average is spent on the data and image transfer over WiFi.

Across all shown devices and apps, migrations required 7.88 seconds to complete on average. This is inclusive of the time required for data transfer in a congested, urban environment, as well as the preparation and checkpoint stages. However, the preparation and checkpoint stages will largely go unnoticed as they occur while the user is presented with the migration target menu and they make their choice. This results in a user-perceived average migration time closer to 5.8 seconds. Given that the data transfer stage is bound by the network bandwidth, it will continually improve as devices and wireless technologies evolve. For example, the latest mobile devices, such as the Google Nexus 5, feature 802.11ac wireless adapters. On 802.11ac capable networks, these devices can significantly outperform the 802.11n performance of the evaluated devices, especially the Nexus 7, which is only capable of operating on the extremely congested 2.4Ghz band. In the future, the data transfer stage could also be greatly reduced by deferring memory transfer using techniques such as post copy supplemented with adaptive pre-paging [26]. This also allows for the data transfer cost to be partially overlapped with the restore and reintegration stages. Looking ahead, to get a better idea of the potential migration times, Figure 14 shows the user-perceived time required for migration excluding data transfer, an average of 1.35 seconds. Note that our prototype is not fully optimized and various migration stages can be improved. For example, Flux currently implements an unoptimized preparation for checkpoint that depends on the Android task idler to stop the app after we have placed it into the background.

Figure 15 shows the average data transferred to migrate each of the apps between devices. We also show the APK size of each app for reference. The amount of data transferred is dominated by the size of the checkpoint image, and in our tests, the compressed data directories sync and record log never exceeded a combined 200KB. None of the migrations required transferring more than 14MB of state during the data transfer stage. Comparing Figure 15 with Figure 12, the migration times are generally correlated with the data transfer sizes. We can loosely say that the larger the app's install size is, the longer it can be expected to take to migrate.

To demonstrate that the recording costs of Flux are modest, we ran the Quadrant Standard [6] and SunSpider [4] benchmarks on both Flux and vanilla Android. Figure 16 shows the results of running the benchmarks on all three types of devices normalized to AOSP and indicates that the overhead is negligible in all cases.

To get a real-world idea of the challenges Flux faces, both in migration performance and support of apps, we analyzed several hundred thousand free Android apps in Google Play. We leveraged PlayDrone [63] to crawl the Google Play store, download the metadata and APKs for a collection of 488,259 apps, and decompiled the APKs to analyze their sources. Since Flux cannot migrate apps which choose to always retain their graphical context, we parsed the sources to identify those that explicitly call Android's `setPreserveEGLContextOnPause`. Of the roughly half million apps we downloaded, this call is only made by 3,300 of them. This indicates that only a small percentage of the apps in Google Play use this feature, and that the Flux approach is expected to work for the vast majority of apps.

Since the cost of pairing devices before migration involves transferring APKs, we also analyzed the collection of apps in Google Play to measure their installation sizes, information included in the metadata associated with each app. To verify that the installation size is a good measure of the actual size of the app APKs, we looked at a random selection of APKs from Google Play and compared their actual size to the installation size. The installation size and actual APK size matched in all cases. Figure 17 shows the cumulative distribution function of all the apps versus their installation size. Roughly 60% of the apps are less than 1 MB in size, and roughly 90% of the apps are less than 10 MB in size.

We also measured the pairing costs for the various devices we used for migration. Pairing consists of a constant data cost component and a cost that scales linearly with the number of installed apps and their install size. The constant data is comprised of a device's system libraries, frameworks and apps. When pairing a Nexus 7 to a Nexus 7 (2013), both running KitKat, the total constant data size that must be synced was 215MB. After accounting for identical files on the target device that can be hard-linked, this is reduced to 123MB. The compressed delta that must be transferred is 56MB.

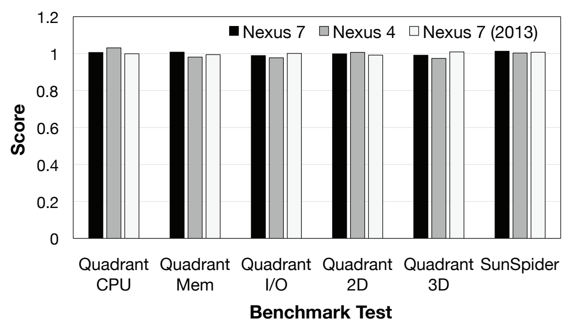

Figure 16. Quadrant Standard and SunSpider benchmark results normalized to AOSP.

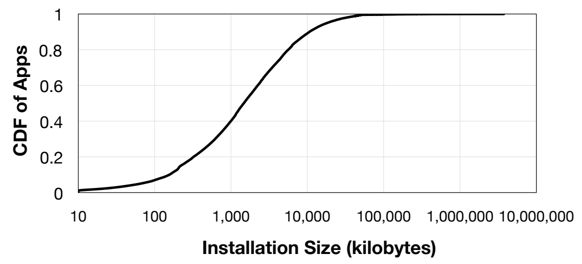

Figure 17. Installation size of Google Play apps.

5. Related Work

Application migration has been extensively studied across a broad range of desktop and server computing systems. Many research operating systems (OSes) have implemented support for process migration, including Accent [54], Amoeba [41], Chorus [56], DEMOS/MP [53], MOSIX [7], Sprite [22], and V [16]. These OSes provide a global namespace and location transparent execution allowing processes to migrate freely across machines. Migrated processes often rely on their home machine for IPC, open files, and system calls, forever tethering them to another machine. None of these approaches are designed for mobile devices, and do not address the key device heterogeneity problems to support migration on mobile devices.

Arguably the most popular migration approach today is VM migration, leveraging virtual machine monitors (VMMs) to virtualize at the hardware level and encapsulate an entire OS [19, 43]. These approaches are used in server and cloud environments, where whole OS virtualization and migration is practical and works well. However, using VMs on mobile devices has been problematic, as existing approaches [10] provide no effective mechanism to enable apps running in VMs to directly leverage hardware device features without substantial performance degradation, especially for apps using 3D accelerated graphics [2, 20]. As a result, no VM-based solutions exist for enabling app migration across commodity smartphones and tablets.

There has been significant research in checkpoint-restore approaches which have been used for migration, spanning the application-level [50, 55, 58], library-level [51, 62], library OS-level [11, 52] and kernel-level [32–34, 39, 48, 49, 60]. None of these approaches work for commodity smartphones and tablets, and do not address the key device heterogeneity problems to support migration on mobile devices. Application-level mechanisms [12, 27], while efficient, are non-transparent, require application-level modifications, and may require nonstandard programming languages [29].

Library checkpoint-restart mechanisms require that applications be compiled or relinked against special libraries. Unlike Flux, such approaches do not capture important parts of the system state, such as interprocess communication and process dependencies through the OS, and do not support significant changes in underlying hardware or the plethora of devices found in mobile platforms.

Library OS approaches encapsulate an entire OS at the user-level to make checkpoint-restore of OS and application state easier across desktop computers, but rely on remote display mechanisms [52], limiting graphics performance. It is unclear how these systems might support app migration across mobile devices. Like distributed OSes, the hard part is migrating across heterogeneous graphics hardware; any library OS attempting to migrate from a desktop to a tablet would need to adopt exactly the kind of mechanisms that are provided by the Flux solution.

Kernel-level approaches include those that require entirely new OSes [32, 39], limiting their deployment, and those that work with commodity OSes such as Linux [33, 48, 49], which led to the current CRIU checkpoint-restore support in Linux [47]. Flux builds on CRIU but specifically targets mobile devices, focusing on providing the necessary hooks to extract and reintegrate application state from Android-specific software device drivers, as well as interactions with system services, and hardware devices to support migration across disparate devices.

Recently, an Android-specific checkpoint-restore project was created for restoring the Android Zygote process for faster booting [1]. However, the project does not support checkpoint-restore of interactive or GUI-based processes and therefore does not support Android apps, supports only same-device checkpoint and restoration, and does not support interaction with hardware devices.

There has also been significant research in record-replay approaches [13, 17, 24, 40, 42, 57, 61], in some cases to even replicate application state across different computers [14]. Unlike Flux, these systems assume a homogeneous environment and are not designed to allow replay with any modifications to the recorded execution.

Other approaches enable replay with varying degrees of modifications from the recorded execution. Crosscut [18] can reduce the information recorded in a log so that, for example, sensitive information can be purged before replay. Scribe [35] replays a recorded application execution until a specified point, and then transitions to live execution instead of replaying the rest of the log. Racepro [36] detects process races due to dependencies in the ordering of system calls by recording an application execution to a log, identifying a pair of system calls that may be racy, truncating the log at the occurrence of the pair of system calls, inverting their order, and replaying the truncated log with the reordered system calls. A few record-replay systems allow new code to be run while replaying a recorded execution [17, 28]. However, this new code cannot have any side effects on the program. More recently, Dora [64] allows transparent mutable replay of application execution even when applications change. Recent work also applies record-replay to graphical contexts by leveraging a record-prune-replay mechanism capable of restoring an OpenGL state by replaying the minimal number of calls necessary [30]. Flux differs from previous approaches in that it targets mobile service invocations and leverages their semantics to guarantee correctness as device state changes, adapts to changes in hardware, and is much lighter weight, making it more suitable for mobile devices.

6. Conclusions and Future Work

Moving computation across glass surfaces has been the vision of science fiction for decades. Recent advances in mobile device computing capacity as well as the proliferation of glass surfaces in cell phones, phablets, tablets, smart TVs, and smart watches, all running similar OSs, will likely enable new forms of computing interactions that extend beyond a single device. We have demonstrated that such experiences are indeed possible with Flux. A user can move apps—mid execution—across Android-based mobile and tablet devices. Our design focused on minimizing the intrusiveness on existing mobile OS stacks and apps. At same time, we wanted to leverage the clean separation between apps and services within a mobile OS to allow for a fluid migration experience where apps can gracefully adapt to changes in hardware devices. We have showed that many popular apps can be migrated without any modifications. In the process, we have fully captured the various overheads in migrating an app. These overheads, while small, are still noticeable. We are exploring various optimizations to mask away these overheads during app migration.

7. Acknowledgments

Nicolas Viennot made possible the analysis of Google Play apps. Donald Porter provided helpful comments on earlier drafts of this paper. This work was supported in part by a Google Research Award, an IBM PhD Fellowship, and NSF grants CNS-1162447, CNS-1422909, and CCF-1162021.

References

[1] 0xlab. Android DMTCP. `http://github.com/0xlab`. Accessed: 2014-04-02.

[2] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 173–187, Cascais, Portugal, Oct. 2011.

[3] Apple Inc. Apple Airplay - Play content from iOS devices on Apple TV. `https://www.apple.com/airplay/`. Accessed: 2015-03-02.

[4] Apple Inc. SunSpider 1.0.2. `https://www.webkit.org/perf/sunspider/sunspider.html`. Accessed: 2014-07-19.

[5] Archana Venkatraman. CIOs Distrust Public Cloud for Mission-Critical Work, says IDC. `http://www.computerweekly.com/news/2240170818/CIOs-distrust-public-cloud-for-mission-critical-work-says-IDC`, Nov. 2012. Accessed: 2015-03-12.

[6] Aurora Software. Quadrant Standard. `http://www.aurorasoftworks.com/products/quadrant`. Accessed: 2014-07-19.

[7] A. Barak and O. Laádan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, Mar. 1998.

[8] R. Baratto, L. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 277–290, Brighton, United Kingdom, Oct. 2005.

[9] R. Baratto, S. Potter, G. Su, , and J. Nieh. MobiDesk: Mobile Virtual Desktop Computing. In *Proceedings of the 10th Annual ACM International Conference on Mobile Computing and Networking (MobiCom 2004)*, pages 1–15, Philadelphia, PA, USA, Sept. 2004.

[10] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *ACM SIGOPS Operating Systems in Review*, 44(4):124–135, Dec. 2010.

[11] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013)*, pages 239–252, Prague, Czech Republic, 2013.

[12] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, June 1997.

[13] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE 2006)*, pages 154–163, Ottawa, Ontario, Canada, June 2006.

[14] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 1–11, Copper Mountain, Colorado, USA, 1995.

[15] Carly Page. Cloud Computing Voted as Fundamentally Insecure. `http://www.theinquirer.net/inquirer/news/2381817/cloud-computing-voted-as-fundamentally-insecure`, Nov. 2014. Accessed: 2015-03-12.

[16] D. R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, Apr. 1984.

[17] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Boston, MA, USA, June 2008.

[18] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage Replay with Crosscut. In *Proceedings of the 6th International Conference on Virtual Execution Environments (VEE 2010)*, pages 13–24, Pittsburgh, Pennsylvania, USA, Mar. 2010.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, pages 273–286, Boston, MA, USA, May 2005.

[20] C. Dall, J. Andrus, A. Van't Hof, O. Laadan, and J. Nieh. The Design, Implementation, and Evaluation of Cells: A Virtual Mobile Smartphone Architecture. *ACM Transactions on Computer Systems (TOCS)*, 30(3):9:1–31, Aug. 2012.

[21] W. R. Dieter and J. E. Lumpp Jr. User-Level Checkpointing for LinuxThreads Programs. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 81–92, Berkeley, CA, USA, June 2001.

[22] F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, Aug. 1991.

[23] Google Inc. Chromecast. `https://www.google.com/chrome/devices/chromecast/`. Accessed: 2014-04-12.

[24] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 193–208, Berkeley, CA, USA, Dec. 2008.

[25] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. *Journal of Physics: Conference Series*, 46:494–499, Sept. 2006.

[26] M. R. Hines and K. Gopalan. Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning. In *Proceedings of the 5th International Conference on Virtual Execution Environments (VEE 2009)*, pages 51–60, Washington, DC, USA, Mar. 2009.

[27] Y. Huang, C. Kintala, and Y. M. Wang. Software Tools and Libraries for Fault Tolerance. *IEEE Bulletin of the Tecnnical Committee on Operating System and Application Environments*, 7(4):5–9, Winter 1995.

[28] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 91–104, Brighton, United Kingdom, Oct. 2005.

[29] L. V. Kale and S. Krishnan. CHARM++: a Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)*, pages 91–108, Washington, DC, USA, Sept. 1993.

[30] S. Kazemi, R. Garg, and G. Cooperman. Transparent Checkpoint-Restart for Hardware-Accelerated 3D Graphics. *CoRR*, abs/1312.6650, Dec. 2013.

[31] J. Kim, R. Baratto, and J. Nieh. pTHINC: A Thin-Client Architecture for Mobile Wireless Web. In *Proceedings of the 15th International World Wide Web Conference (WWW 2006)*, pages 143–152, Edinburgh, Scotland, May 2006.

[32] B. A. Kingsbury and J. T. Kline. Job and Process Recovery in a UNIX-based Operating System. In *Proceedings of the 1989 USENIX Winter Technical Conference*, pages 355–364, San Diego, CA, USA, Jan. 1989.

[33] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 323–336, Santa Clara, CA, USA, June 2007.

[34] O. Laadan, D. Phung, and J. Nieh. Transparent Checkpoint-Restart of Distributed Applications on Commodity Clusters. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing (Cluster 2005)*, pages 1–13, Boston, MA, USA, Sept. 2005.

[35] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2010)*, pages 155–166, New York, NY, USA, June 2010.

[36] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, , and J. Nieh. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 353–367, Cascais, Portugal, Oct. 2011.

[37] A. Lai and J. Nieh. Limits of Wide-Area Thin-Client Computing. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, pages 228–239, Marina del Rey, CA, USA, June 2002.

[38] A. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya. Improving Web Browsing on Wireless PDAs Using Thin-Client Computing. In *Proceedings of the 13th International World Wide Web Conference (WWW 2004)*, pages 143–154, New York, NY, May 2004.

[39] C. R. Landau. The Checkpoint Mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 86–91, Dourdan, France, Sept. 1992.

[40] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4), Apr. 1987.

[41] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[42] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 22–31, San Diego, California, USA, June 2007.

[43] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 25–25, Anaheim, CA, USA, Apr. 2005.

[44] J. Nieh, S. J. Yang, and N. Novik. Measuring Thin-Client Performance Using Slow-Motion Benchmarking. *ACM Transactions on Computer Systems (TOCS)*, 21(1):87–115, Feb. 2003.

[45] J. Onorato. Modifying AIDL Compiler to Generate C++ Code. `https://groups.google.com/forum/#!topic/android-framework/i0SWc9cHEJ0`, Nov. 2011. Accessed: 2015-03-12.

[46] OpenSignal. Android Fragmentation Visualized. `http://opensignal.com/reports/2014/android-fragmentation/`, Aug. 2014. Accessed: 2015-03-11.

[47] OpenVZ. Checkpoint/Restore in Userspace. `http://www.criu.org`. Accessed: 2014-03-14.

[48] OpenVZ Linux Containers. `http://www.openvz.org`. Accessed: 2014-03-15.

[49] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, Boston, MA, USA, Dec. 2002.

[50] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Dept. of Computer Science, University of Tennessee, July 1997.

[51] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 213–223, New Orleans, LA, USA, Jan. 1995.

[52] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 291–304, Newport Beach, CA, USA, Mar. 2011.

[53] M. L. Powell and B. P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP 1983)*, pages 110–119, Bretton Woods, NH, USA, Oct. 1983.

[54] R. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*

(SOSP 1981), pages 64–75, Pacific Grove, California, USA, Dec. 1981.

[55] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, July 2002.

[56] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus Distributed Operating System. *Computing Systems*, 1(4):305–370, 1988.

[57] Y. Saito. Jockey: a User-Space Library for Record-Replay Debugging. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG 2005)*, pages 69–76, Monterey, CA, USA, Sept. 2005.

[58] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05) - Workshop 18*, page 300.2, Washington, DC, USA, Apr. 2005.

[59] SC Magazine. 2013 Mobile Device Survey. `http://www.scmagazine.com/2013-mobile-device-survey/slideshow/1222/`, 2013. Accessed: 2014-04-20.

[60] B. K. Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Stanford University, Aug. 2000.

[61] D. Subhraveti and J. Nieh. Record and Transplay: Partial Checkpointing for Replay Debugging Across Heterogeneous Systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2011)*, pages 109–120, San Jose, CA, USA, June 2011.

[62] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, 20(227):40–48, Feb. 1995.

[63] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2014)*, pages 221–233, Austin, TX, USA, June 2014.

[64] N. Viennot, S. Nair, and J. Nieh. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 127–138, Houston, TX, USA, Mar. 2013.

[65] S. J. Yang, J. Nieh, S. Krishnappa, A. Mohla, and M. Sajjadpour. Web Browsing Performance of Wireless Thin-Client Computing. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, pages 68–79, Budapest, Hungary, May 2003.

[66] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari. The Performance of Remote Display Mechanisms for Thin-Client Computing. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 131–146, Monterey, CA, USA, June 2002.