Study and Refactoring of Android Asynchronous Programming

Yu Lin, Semih Okur
Computer Science Department
University of Illinois at Urbana-Champaign, USA
{yulin2, okur2}@illinois.edu

Danny Dig
School of Electrical Engineering and Computer Science
Oregon State University
digd@eecs.oregonstate.edu

***Abstract*—To avoid unresponsiveness, a core part of mobile development is asynchronous programming. Android provides several async constructs that developers can use. However, developers can still use the inappropriate async constructs, which result in memory leaks, lost results, and wasted energy. Fortunately, refactoring tools can eliminate these problems by transforming async code to use the appropriate constructs.**

In this paper we conducted a formative study on a corpus of 611 widely-used Android apps to map the asynchronous landscape of Android apps, understand how developers retrofit asynchrony, and learn about barriers encountered by developers. Based on this study, we designed, implemented, and evaluated ASYNCDROID, a refactoring tool which enables Android developers to transform existing improperly-used async constructs into correct constructs. Our empirical evaluation shows that ASYNCDROID is applicable, accurate, and saves developers effort. We submitted 45 refactoring patches, and developers consider that the refactorings are useful.

I. INTRODUCTION

According to a Gartner report [9], by 2016 more than 300 billion apps for smartphones and tablets will be downloaded annually. Android is the dominant platform, by 4x over the next largest platform and 1.7x over all the others combined [16]. Previous studies [37], [50] have shown that unresponsiveness is the number one bug that plagues Android apps. To avoid unresponsiveness, app developers use asynchronous programming to execute CPU-bound or blocking I/O operations (e.g., accessing the cloud, database, filesystem) in background and inform the app when the result becomes available.

In order to make asynchronous programming easier, Android provides three major async constructs: `AsyncTask`, `IntentService` and `AsyncTaskLoader`. `AsyncTask` is designed for encapsulating short-running tasks while the other two are good choices for long-running tasks. However, as our formative study on a corpus of 611 shows, `AsyncTask` is the most widely used construct, dominating by a factor of 3x over the other two choices combined.

If improperly used, `AsyncTask` can lead to memory leaks, lost results, and wasted energy. Our previous study [36] found that developers usually hold a reference to a GUI component in an `AsyncTask`, so that they can easily update GUI based on task results. However, there are several instances in which the Android system can destroy and recreate a GUI component while an `AsyncTask` is running: when a user changes the screen orientation, or navigates to another screen on the same app, switches to another app, clicks the “Home” button and navigates back, etc. If the `AsyncTask` is still running and it holds GUI reference, the destroyed GUI cannot be garbage collected, which leads to memory leaks.

On the other hand, if an `AsyncTask` that finished its job updates a GUI component that has already been destroyed and recreated, the update is sent to the destroyed GUI rather than the recreated new one, and cannot be seen by the user. Thus, the task result is lost, frustrating the user. Moreover, the device wasted its energy to execute a task whose result is never used. As pointed out by many forums [1], [4], [6], this problem is widespread and is critical for long-running tasks. `IntentService` and `AsyncTaskLoader` do not have the above limitations because they do not hold a reference to GUI and instead use a radically different mechanism to communicate with the GUI. To avoid the above problems of `AsyncTask`, developers must refactor `AsyncTask` code into enhanced async constructs such as `IntentService`.

However, manually applying this refactoring is non-trivial due to drastic changes in communication with GUI. This is a challenging problem because a developer needs to transform a shared-memory based communication (through access to the same variables) into a distributed-style (through marshaling objects on special channels). First, the developer needs to determine which objects should flow into or escape from `IntentService`. Unlike `AsyncTask`, the objects that flow into or escape from `IntentService` should be serializable. Determining this requires tracing the call graph and type hierarchy. Second, the developer needs to rewire the channels of communication. Whereas `AsyncTask` provides handy event handlers for callback communication, `IntentService` does not. `IntentService` and GUI can only communicate through sending and receiving broadcast messages. This requires developer to replace event handlers with semantic-equivalent broadcast receivers, which is tedious. Third, the developer needs to infer where to register the broadcast receivers. Registering at inappropriate places can still lead to losing task results.

In this paper we first present a formative study to understand how developers use different Android async constructs. We analyzed a corpus of 611 most popular open-source Android apps, comprising 6.47M SLOC. To further put the study results in a broader context, we then surveyed 10 expert Android developers. The formative study answers the following questions:

RQ1: *How do Android developers use asynchronous pro-*

gramming? Mapping the landscape of usage of async constructs in Android is useful for researchers, library designers, and is educational for developers. We found that 161 (32%) of the studied apps use at least one asynchronous programming, resulting in 1893 instances. Out of these, `AsyncTask` is the most widely used.

RQ2: *How do Android developers retrofit asynchrony into existing apps?* Must asynchrony be designed into a program, or can it be retrofitted later? What are the most common transformations to retrofit asynchrony? Answering this question is important for software evolution researchers, as well as tool builders. We found widespread use of refactorings, both from sequential code to async, and from basic async to enhanced async. We found the following code evolution scenario: developers first convert sequential code to `AsyncTask`, and those that continue to evolve the code for better use of asynchrony refactor it further into enhanced constructs.

RQ3: *How do expert developers interpret the disparity in usage of async constructs?* Answering this question can provide educational value for developers. We found that experts think `AsyncTask` is being overused at the expense of other enhanced async constructs, and many inexperienced Android developers do not know its problem. They also suggest `AsyncTask` should only be considered for short-running tasks (i.e., less than a second). They suggest that the current guides and examples of `IntentService` are not enough, and point out the need for refactoring tools to help unexperienced developers use and learn about the enhanced async constructs and the different style of communicating with the GUI.

Inspired by the results of our formative study, we designed, developed, and evaluated ASYNCDROID, an automated refactoring tool that transforms shared-memory into distributed-style communication in the context of Android async programming. ASYNCDROID refactors `AsyncTask` into `IntentService`. We developed ASYNCDROID as an Eclipse plugin, thus it offers all the convenience of a modern refactoring tool: it enables the user to preview and undo changes and it preserves formatting and comments. To use it the programmer only needs to select an `AsyncTask` instance, then ASYNCDROID verifies that the transformation is safe, and rewrites the code if the preconditions are met. However, if a precondition fails, it warns the programmer and provides useful information that helps the programmer fix the problem.

This paper makes the following contributions:

Problem Description: To the best of our knowledge, we are the first to propose solving the generic problem of converting shared-memory into distributed-style communication through refactoring. We do this in the context of Android async programming.

Formative study: To the best of our knowledge, this paper presents the first quantitative and qualitative study to (i) map the asynchronous landscape of Android, (ii) understand how developers retrofit asynchrony, and (iii) learn about barriers encountered by developers.

Algorithms: We designed the analysis and transformation algorithms to address the challenges of refactoring from shared-memory communication (as used in `AsyncTask`) to distributed-style communication (as used in `IntentService`). The algorithm determines the incoming and outgoing objects in/from `IntentService`, replaces event handlers with broadcast messages and receivers, and infers where to register broadcast receivers.

Tool: We implemented the algorithms in ASYNCDROID, a refactoring tool built on top of Eclipse refactoring engine.

Evaluation: We evaluated ASYNCDROID empirically, by refactoring 97 `AsyncTasks` in 9 popular open-source Android projects. We evaluate ASYNCDROID from three aspects. First, 45% of the `AsyncTasks` pass the refactoring preconditions, and with minor manual changes another 10% `AsyncTasks` can pass preconditions. This means the refactoring is highly applicable. Second, ASYNCDROID changed 3386 SLOC in 77 files in total, determined that 148 variables flow into or escape from `IntentService`, moved 14 methods into `IntentService`, and marked 18 types as serializable. This task is very large and challenging to be performed manually, but ASYNCDROID performs each refactoring in a few seconds. This shows that ASYNCDROID can save developer effort. Third, we submitted 45 refactoring patches in 7 projects. 4 projects replied and considered our changes to be correct, and they accepted 15 refactorings. This shows ASYNCDROID is valuable.

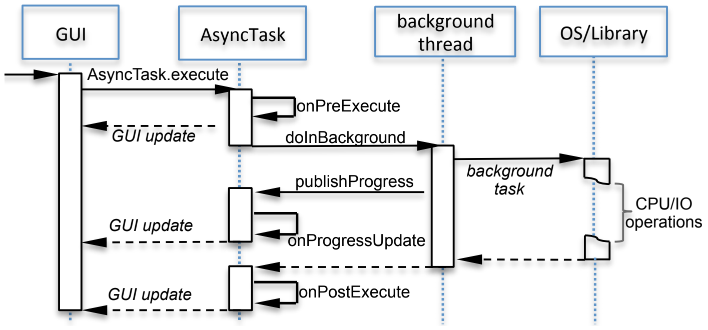
The tool and subjects for our study and evaluation are available at: http://web.engr.illinois.edu/~yulin2/asyncdroid/

II. BACKGROUND ON ANDROID

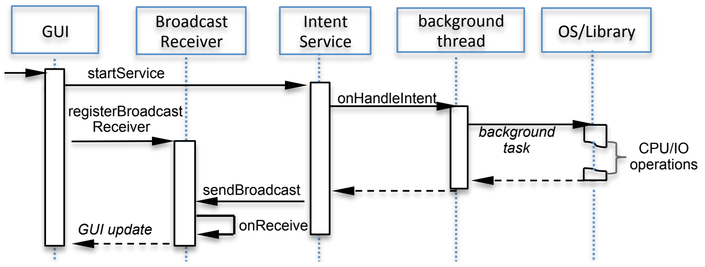
An Android app consists of four types of components: *activity*, *service*, *broadcast receiver* and *content provider*. In this paper, we focus on the first three components. *Activities* contain GUI widgets and represent the GUI screens that are shown to users. *Services* do not provide GUI but perform invisible operations (e.g., playing music). *Broadcast receivers* provide a channel for *activities* and *services* to communicate.

Similar to many other GUI frameworks such as Swing [13] and SWT [17], Android uses an event-driven model. Events in Android include lifecycle events (e.g., activity creation), user actions (e.g., button click, menu selection), sensor inputs (e.g., GPS, orientation change), etc. Developers define event handlers to respond to these events. For example, `onCreate` handler is a lifecycle event handler and is invoked automatically by the OS when an activity creation event occurs, while `onClick` handler of a button is invoked when user clicks the button.

Android framework uses a **single thread model** to process events [3]. When an application is launched, the system creates a *UI thread*, in which it will run the application. This thread is in charge of dispatching UI events to appropriate widgets or lifecycle events to activities. It puts events into a single event queue, dequeues events, and executes event handlers. By default, all the event handlers defined in the above three components are processed by the UI thread one by one. Thus, if any handler executes CPU or IO bound operations such as network access, the UI thread will be blocked and no further events can be processed. This will lead to unresponsiveness. To avoid unresponsiveness, developers should exploit asynchrony and schedule long-running tasks on background threads. Android provides three major async constructs: `AsyncTask`, `IntentService` and `AsyncTaskLoader`.



(a) Where is `AsyncTask` code executing?



(b) Where is `IntentService` code executing?

Fig. 1: Flow of Android async constructs

A. *AsyncTask*

`AsyncTask` provides a `doInBackground` method for encapsulating asynchronous work. It also provides four event handlers which are run in the UI thread. The `doInBackground` and these event handlers share variables through which the background task can communicate with UI. Figure 1(a) shows the flow of `AsyncTask`. An `AsyncTask` is started by invoking `execute` method in UI thread. UI thread first executes `onPreExecute` handler. Then `doInBackground` method is executed in a *background thread*. While the task is executing, it can report its progress to the UI thread by invoking `publishProgress` and implementing `onProgressUpdate` handler. Finally, UI thread executes the `onPostExecute` handler after `doInBackground` finishes. `onPostExecute` takes the task result as its parameter. Notice that user can also cancel an `AsyncTask`. If a task is canceled, `onCancel` handler will be executed instead of `onPostExecute`.

Figure 2(a) shows a real-world example from *owncloud-android* app, which is an Android client for a cloud server. `LogActivity` starts a `LoadingLogTask` at line 11. The task reads some files in `doInBackground` (lines 31, 32) and returns a string (line 35). `onPostExecute` takes this string and uses it to update a text view (line 21). Notice that `LoadingLogTask` is declared as a non-static inner class, so it holds a reference to `LogActivity`.

Though `AsyncTask` is an easy-to-use construct, it is ideally used for short tasks. Otherwise, the three problems introduced in Sec. I (memory leaks, lost results, wasted energy) can occur.

B. *IntentService*

`IntentService` belongs to the *Service* component, but its encapsulated operations are executed in a background thread instead of UI thread. Figure 1(b) shows the flow of `IntentService`. Service is started when `startService` is invoked. Then `onHandleIntent` method is executed in a background thread. Unlike `AsyncTask`, `IntentService` uses a distributed-style programming to communicate with UI thread. To get the task result, the GUI that starts the service should register a broadcast receiver. After the task is finished, `IntentService` sends its task result via `sendBroadcast` method. Once the registered receiver on GUI receives this broadcast, its `onReceive` method will be executed on UI thread, so it can get the task result and update GUI.

Figure 2(b) shows an equivalent implementation of `LogActivity` using `IntentService`. The background task from Fig. 2(a) is now put into `onHandleIntent` method (line 31). The result is wrapped by an `Intent` object, which is marshalled and sent via broadcast (lines 34, 35). `Intent` is analogous to a hash map whose key is a string and value is a serializable object (i.e., implements `java.io.Serializable` or `android.os.Parcelable`). It is the only medium through which different components (e.g., *service* and *broadcast receiver*) can exchange data. Finally, `onReceive` unwraps the result and updates the text view (line 21). Notice that to register a receiver, the developer should provide a filter (represented by a string) to specify which broadcast the receiver can receive. For example, line 2 defines a filter “FILTER”. Lines 8 and 29 use it to register receiver and send broadcast.

Since `IntentService` is not affected by the destruction of the GUI objects that started it, it does not suffer from the problems introduced in Sec. I. Thus, it can be safely used for both short or long tasks.

C. *AsyncTaskLoader*

`AsyncTaskLoader` is built on top of `AsyncTask`, and it provides similar handlers as `AsyncTask`. Unlike `AsyncTask`, `AsyncTaskLoader` is lifecycle aware: Android system binds/unbinds the background task with GUI according to GUI’s lifecycle. Thus, it can also solve the problems we mentioned in Sec. I. However, `AsyncTaskLoader` is introduced after Android 3.0, and it only supports two GUI components: *activity* and *fragment*.

III. Formative Study of Android Asynchrony

We want to assess the state of practice async programming in open-source Android apps. To obtain a deep understanding of asynchronous programming practices in Android, we answer three research questions. We now answer each of these questions, by first presenting the methodology and corpus, and then the results.

RQ1: How do Android developers use asynchronous programming?

To answer this question, we studied *all* async constructs provided by the standard Android libraries: `AsyncTask`, `IntentService`, `AsyncTaskLoader` and also the legacy-style Java `Thread`.

```java
public class LogActivity extends Activity {


   private String mLogPath = FileStorageUtils.getLogPath();
   protected void onCreate() {
      ...
      LoadingLogTask task = new LoadingLogTask(view);



      task.execute();
   }
   private class LoadingLogTask extends AsyncTask {
      private TextView textView;
      private Exception exception;
      public LoadingLogTask(TextView view){ textView = view; }
      public void onPostExecute(String result) {


         if (exception == null) {
            textView.setText(result);
         } else Log.i(exception.getMessage());
      }



      public String doInBackground(String[] args) {


         try {
            File file = new File(mLogPath);
            ...


            return text;
         } catch (Exception e) {
            exception = e;


            return null;
}}}}
```

(a) Using `AsyncTask`

```java
public class LogActivity extends Activity {
   public static final FILTER = "LogActivity_receiver";
   private LoadingLogReceiver receiver;
   private String mLogPath = FileStorageUtils.getLogPath();
   protected void onCreate() {
      ...
      receiver = new LoadingLogReceiver(view);
      this.registerReceiver(receiver, new IntentFilter(FILTER));
      Intent intent = new Intent(this, LoadingLogService.class);
      intent.putExtra("mLogPath", mLogPath);
      this.startService(intent);
  }
  private class LoadingLogReceiver extends BroadcastReceiver {
     private TextView textView;
     private Exception exception;
     public LoadingLogReceiver(TextView view){ textView = view; }
     public void onReceive(Context context, Intent intent) {
        String text = intent.getStringExtra("RV");
        Exception exception = (Exception)intent.getSerializableExtra("exception");
        if (exception == null) {
           textView.setText(text);
        } else Log.i(exception.getMessage());
}}}
public class LoadingLogService extends IntentService {
   private String mLogPath;
   private Exception exception;
   public void onHandleIntent(Intent intent) {
      this.mLogPath = intent.getStringExtra("mLogPath");
      Intent result = new Intent(LogActivity.FILTER);
      try {
         File file = new File(mLogPath);
         ...
         result.putExtra("exception", exception);
         result.putExtra("RV", text);
         sendBroadcast(result);
      } catch (Exception e) {
         exception = e;
         result.putExtra("exception", exception);
         result.putExtra("RV", null);
         sendBroadcast(result);
}}}
```

(b) Using `IntentService`

Fig. 2: Asynchronous code from *owncloud-android* app. `onCreate` method starts a background task to read a log file, and the result is shown in a `TextView`. (a) and (b) shows two semantic-equivalent implementations using `AsyncTask` and `IntentService`, respectively.

Corpus-1. To collect representative Android apps, we chose Github [11]. To distinguish Android apps in Github, we first searched all apps whose `README` file contains the "Android" keyword. Because we want to analyze recently updated apps, we filtered for apps which have been modified at least once since July 2014. Then, we sorted all these apps based on their star count and gathered the top 500 apps. To make sure that these apps have Android projects, we check whether every app has at least one *"AndroidManifest.xml"* file, which every Android project must contain in its root directory. After all filters, our code corpus has 500 Android apps, comprising 4.69M non-blank, non-comment SLOC (as reported by SLOCCount [14]).

Methodology. We built a tool, ASYNCANALYZER to automatically analyze the usage of async constructs in our corpus. ASYNCANALYZER uses Eclipse API for syntactic analysis. It builds an abstract syntax tree (AST) for each source file and traverses ASTs to gather the usage statistics for the four async constructs. Doing the analysis at the level of ASTs and not at the textual level, improves the accuracy in several ways.

TABLE I: Usage of async constructs in the Corpus-1

	# Instances	# App	App%
`AsyncTask`	938	97	19%
`Thread`	655	110	22%
`IntentService`	182	30	6%
`AsyncTaskLoader`	118	14	3%

First, it is immune to noise generated by text that matches names of the async constructs (e.g., import statements, comments, variable names, etc.) but does not represent an instance of an async construct. Second, it correctly accounts for the ways in which developers instantiate async constructs: via anonymous inner classes (e.g., `myAsyncTask = new AsyncTask(...)`) and via class subtyping (e.g., `class MyService extends IntentService`.

Results. Table I tabulates the usage statistics of async constructs. The three columns show the total number of construct instances, the total number of apps with instances of the construct, and the percentage of apps with instances of the construct.

As we see from the table, `AsyncTask` is the most popular async construct in our corpus, based on the total number of instances. However, if we count the total number of apps that use at least one of these constructs, then the legacy-style `Thread` is the most popular, despite the fact that Android already provides three special async constructs. `AsyncTaskLoader` and `IntentService` are not as popular as the other two.

RQ2: How do Android developers retrofit asynchrony into existing apps?

Next we analyze how developers introduce async constructs into their apps. We want to determine whether developers introduced async constructs when (i) implementing new features (i.e., asynchrony was added to a new program element when writing code from scratch), (ii) refactoring from existing

TABLE II: How developers introduce `AsyncTask` and `IntentService` in the Corpus-2

Type	# Instances
Newly added `AsyncTask`	277
Refactor sequential code to `AsyncTask`	103
Refactor `Thread` to `AsyncTask`	18
Newly added `IntentService`	205
Refactor sequential code to `IntentService`	13
Refactor `Thread` to `IntentService`	18
Refactor `AsyncTask` to `IntentService`	9
Refactor `AsyncTaskLoader` to `IntentService`	5

TABLE III: How developers introduce `AsyncTask` and `AsyncTaskLoader` in the Corpus-3

Type	# Instances
Newly added `AsyncTask`	73
Refactor sequential code to `AsyncTask`	24
Refactor `Thread` to `AsyncTask`	2
Newly added `AsyncTaskLoader`	15
Refactor sequential code to `AsyncTaskLoader`	3
Refactor `AsyncTask` to `AsyncTaskLoader`	10
Refactor `Thread` to `AsyncTaskLoader`	3

sequential code, (iii) refactoring from another existing async construct.

In order to be able to detect transitions between basic and enhanced async constructs, we need to find projects where developers are aware that the enhanced constructs exist. Thus, we use 2 different corpora to study `IntentService` and `AsyncTaskLoader`, respectively:

Corpus-2. We collected 93 random open-source Android apps from Github, comprising 1.54M SLOC, which use both `AsyncTask` and `IntentService` constructs in their latest snapshot.

Corpus-3. We collected 18 random open-source Android apps, comprising 0.24M SLOC, which use both `AsyncTask` and `AsyncTaskLoader` constructs in their latest snapshot.

Methodology. In order to identify transitions, we study not only the latest version of the code, but also the first version where developers introduce async constructs. To do this, we automatically searched the commit history of our corpora through Gitective API [10], identified the commits that add import statements to `AsyncTask`, `AsyncTaskLoader`, or `IntentService`. After automatically finding commits that introduce these async constructs, we manually inspected the versions before and after such commits in order to understand how these async constructs are introduced.

Results. Tables II and III show how `AsyncTask`, `IntentService`, and `AsyncTaskLoader` are introduced.

The results show that in many cases developers refactor sequential code to `AsyncTask`. This observation confirms our previous findings [36]. However, the refactorings for `IntentService` and `AsyncTaskLoader` mostly come from other async constructs. This shows the following code evolution scenario: developers first convert sequential code to `AsyncTask`, and those that continue to evolve the code for better use of asynchrony refactor it further into `IntentService` or `AsyncTaskLoader`.

RQ3: How do expert developers interpret the disparity in usage of async constructs?

To shed light into this question, we conducted a survey with expert Android developers.

Methodology. To find expert developers, we used StackOverflow [15], which is the pioneering Q&A website for programming. In StackOverflow, users are sorted by their points that they received from their answers for questions which are associated with some tags. We contacted the top 10 users for the "android-async" tag and these 10 people are the ones who answered the questions related to Android async programming most. On average, each of them answered 2095 questions on StackOverflow. We got replies from 5 of them, including the author of a popular Android programming book [40].

Results. We asked three questions and summarized the experts' answers below:

Q1) Why are there still lots of legacy-style `Thread` uses even though Android provides three dedicated async constructs?

First, `Thread` construct has been around since the beginning and many Android developers formerly developed Java apps. Developers are very familiar with `Thread` and they do not have time to learn something new, thus they continue using it. Second, `Thread` is suitable for other scenarios, such as parallelism and scheduled tasks.

Q2) Why are there many more `AsyncTask` uses than the other two enhanced constructs even though `AsyncTask` may lead to memory leaks, lost task results, and wasted energy?

They all agree that `AsyncTask` "*is being overused*" at the expense of the other two enhanced constructs. As a main reason, they invoke a historical account "*`AsyncTask` was advertised to the developers a lot in Android documentation*" and it "*got a lot of good press early on*". On the other hand, they thought "*many developers coming from desktop do not realize the async nature of the Android memory management*". They also mention that `AsyncTaskLoader` has been around only for a short time and is harder to implement, and Google has not provided production-level examples of code that use `IntentService` and `AsyncTaskLoader`.

As a guidance on Android async programming, they suggest that `AsyncTask` or `Thread` should only be considered for short tasks (i.e., less than one second). For the work that will take more than a second, developers should use `IntentService`.

Q3) Do you think that developers can benefit from a refactoring tool from `AsyncTask` to `IntentService`?

They concluded that the technical challenges make the automation really hard: "*it would be a very difficult task, so the end solution may appear very complicated*", "*that would be quite a challenge to do automatically*". One also said that "*it may help beginner, but senior developers still like using their preferred way of writing async*" and another said "*it would have to be very compelling for users to take their existing code* *and change it - especially to something they do not already understand*".

IV. REFACTORING ASYNCTASK TO INTENTSERVICE

Based on our findings from Sec. III, developers tend to choose `AsyncTask` for Android async programming. However, `AsyncTask` is not fit for long-running tasks, where developers should use `IntentService` or `AsyncTaskLoader`. Inspired by these findings, we propose ASYNCDROID, an automated refactoring tool that transforms shared-memory into distributed-style communication in the context of Android async programming. ASYNCDROID refactors existing `AsyncTask` into `IntentService`. We have implemented ASYNCDROID as a plugin in the Eclipse IDE, on top of Eclipse JDT [8] and refactoring engine [7]. To use ASYNCDROID, the developer selects the `doInBackground` method in the `AsyncTask` she wants to transform, and then chooses CONVERT TO INTENTSERVICE option from the refactoring menu. By looking at transformations performed (e.g., using Eclipse's preview refactoring feature), developers can educate themselves on how to transform between `AsyncTasks` to `IntentServices`.

A. Refactoring Challenges

There are three main challenges that make it hard to execute the refactoring quick and flawlessly by hand. First, the developer needs to determine which objects should be transferred into `IntentService` and `BroadcastReceiver`, and how to transfer them. Second, the developer should establish channels to enable communication between `IntentService` and GUI. Third, the developer must register the receiver properly in order to receive the computation result from the established channel.

Transfer Objects from/to `IntentService`. As shown in Sec. II, the non-local objects required by `AsyncTask` are passed as method arguments or can be directly accessed as fields from the outer class. However, the objects that flow into and escape from `IntentService` have to be wrapped and sent via an `Intent` object. Similar to distributed-style programming, objects transferred in this way are required to be serializable [2].

For example, at line 31 in Fig. 2(a), field `mLogPath` flows into `doInBackground` method. Thus, it should be transferred to `IntentService` during refactoring (line 10 in Fig. 2(b)). Determining objects transfer requires a nontrivial inter-procedural analysis of the code: the developer must trace (i) the call graph to figure out which objects flow into `IntentService` and `BroadcastReceiver`, and (ii) type hierarchy to check if the objects can be serialized.

Establish Channels for Communication. `AsyncTask` provides four handlers that enable developers to interact with GUI. Background task and handlers exchange data by accessing the shared memory. However, `IntentService` sends broadcast to `BroadcastReceiver` to communicate with GUI. Thus, the developer needs to rewire the channels of communication. To achieve this, one must split `AsyncTask` into `IntentService` and `BroadcastReceiver`. Splitting an `AsyncTask` includes moving the related fields and methods into `IntentService` and `BroadcastReceiver`, which requires to trace the call graph. Additionally, the developer has to write extra code for sending broadcast, which makes the refactoring more tedious.

Where to Register the Receiver. In order to receive the computation result from the channels, the GUI needs to register a `BroadcastReceiver`. A naive approach is to register at the original call site where the `AsyncTask` is created. For example, in Fig. 2(a), the task is created in `onCreate` at line 7. While using `IntentService` in Fig. 2(b), the receiver is registered at the same place in `onCreate` (line 8). This approach only works when the original call site is already in a lifecycle event handler, such as `onCreate` method. Lifecycle events are guaranteed to be triggered by OS during GUI recreation, so the receiver can be registered automatically.

However, if the call site is not in a lifecycle event handler, the receiver cannot be registered unless the event is triggered again after GUI recreation. For example, the call site can be in the `onClick` listener of a button, so the receiver can only be registered when the user clicks the button. If the GUI containing the button is recreated while the background task is running, the recreated GUI cannot receive the broadcast unless the button is clicked again. Thus, the developer also needs to infer where to register `BroadcastReceiver` during the refactoring. This is a non-trivial insight for developers, because online documents only show basic Android asynchronous programming scenarios, where this is not a concern.

B. The Canonical Form of `AsyncTask` Code and Refactoring Preconditions

A key insight in designing refactorings is that there is a canonical form for input code [30], [42]. This canonical form adheres to the preconditions of the refactoring, so that the result of the transformation is indeed correct. This means that if the input code is not in canonical form, it is necessary to transform into canonical form before performing the refactoring. ASYNCDROID has several preconditions, which together dictate the canonical form which the source code must adhere to for a successful transformation:

P1: *All variables that flow into or escape from `doInBackground` are or can be marked as serializable.* This is required because such variables need to be transferred to `IntentService` and `BroadcastReceiver` via marshalling/unmarshalling.

P2: *All the methods invoked in `doInBackground` should also be accessible by `IntentService`.* Because most AsyncTasks are declared as non-static inner classes, they can call methods from the outer class. Since IntentService is not an inner class, it needs to be able to call the same set of methods from the outer class, so their visibility needs to be appropriate.

P3: *The refactored task is directly extended from `AsyncTask` and is not subclassed.* This precondition prevents the refactoring from breaking the inheritance relation and affecting other `AsyncTasks` that are not refactored in the type hierarchy.

P4: *An `AsyncTask` instance is only used when invoking `AsyncTask.execute`.* For example, if a task instance is used as a method argument or return value, ASYNCDROID halts the refactoring to avoid changing the design contract of the method. On the other hand, `AsyncTask` defines some methods that are not supported by `IntentService` (e.g., `AsyncTask.cancel`). If these methods are invoked on the task instance, AsyncDroid halts the refactoring.

Figure 2(a) is a valid example of a target program that meets all these preconditions, thus it can be refactored by AsyncDroid. We will see in Sec. V how many real-world programs readily meet these preconditions.

C. The Refactoring Algorithm

AsyncDroid takes the following steps to refactor an `AsycnTask` to `IntentService`.

Analyzing Transferred Objects. Since `doInBackground` and `onHandleIntent` are semantic-equivalent methods that enclose background task, AsyncDroid needs to analyze `doInbackground` to determine which objects should be transferred.

We define the *Target Class*, *Incoming Variables* and *Outgoing Variables* for `IntentService` as following:

Definition 1 (Target Class ($\mathcal{TC}$)): The top-level or static inner class that creates and starts the `AsyncTask`.

Definition 2 (Incoming Variables ($\mathcal{IV}$)): The set of non-local variables flow into `doInBackground`, which have to be transferred to `IntentService`, is:

$F_{TC} \cup F_{task} \cup Args_{task} \cup LV_{TCM}$ where:

- F_{TC} is the set of $\mathcal{TC}$'s fields when the `AsyncTask` is a non-static inner class of $\mathcal{TC}$
- F_{task} is the set of `AsyncTask`'s fields that are initialized in its constructors or `onPreExecute` handler
- $Args_{task}$ are the arguments of `AsyncTask.execute` method
- LV_{TCM} is the set of ***final*** local variables declared in the $\mathcal{TC}$'s method where the task is created, when the `AsyncTask` is an anonymous inner class of $\mathcal{TC}$

Notice that collecting $\mathcal{IV}$ requires inter-procedural analysis since `doInBackground` may invoke other methods defined in $\mathcal{TC}$ or the `AsyncTask`.

Definition 3 (Outgoing Variables ($\mathcal{OV}$)): The set of variables that escape from `doInBackground` and need to be transferred from `IntentService` to `BroadcastReceiver`, is:

$F^M_{TC} \cup F^M_{task} \cup RV$ where:

- F^M_{TC} is the set of $\mathcal{TC}$'s fields that are modified in `doInBackground` method
- F^M_{task} is the set of `AsyncTask`'s fields that are modified in `doInBackground` method and used in `onPostExecute`
- RV is the return value of `doInBackground` method

An example of $\mathcal{OV}$ is the return value `text` and a field `exception` in Fig. 2(a) (lines 35, 37). F^M_{TC} and F^M_{task} are $\mathcal{OV}$ because `IntentService` and GUI holds and operates on different copies of objects due to (de)serialization. They should be written back in `BroadcastReceiver` otherwise the modifications are lost. Note that there is no way to write back modified incoming LV_{TCM} or $Args_{task}$ due to the communication mechanism of `IntentService`. However, we never find a case in practice where these two types of $\mathcal{IV}$ are modified. A reasonable explanation is that modifying them in a background thread can introduce data races.

As mentioned in Sec. IV-A, the objects in the $\mathcal{IV}$ and $\mathcal{OV}$ set are required to be serializable. AsyncDroid traverses the type hierarchy and checks the serializability of an variable type based on its definition [12]: a type is serializable if it implements `java.io.Serializable` and all of its fields' types are serializable, unless the field is ***transient***. AsyncDroid also refactors a type to implement `java.io.Serializable` if its fields conform to the definition but it has not implemented yet. Note that when refactoring a type to be serializable, AsyncDroid also checks the serializability of all its subtypes to guarantee the transformation is safe, and it only refactors the type defined in source code, not libraries. If any of the $\mathcal{IV}$ and $\mathcal{OV}$ are not serializable or cannot be refactored, the refactoring fails precondition **P1**.

Generating and Starting `IntentService`. First, for the target `AsyncTask`, AsyncDroid creates a corresponding `IntentService` class. The method body of `onHandleIntent` is moved from `doInBackground`. Unlike `AsyncTask`, `IntentService` class cannot be a non-static inner class. Thus, AsyncDroid creates it as an independent class. AsyncDroid creates a field for each variable in $\mathcal{IV}$ and $\mathcal{OV}$. AsyncDroid adds statements to unwrap objects in $\mathcal{IV}$ from `Intent` at the beginning of `onHandleIntent` (line 28 in Fig. 2(b)) . At every exit of `doInBackground`, AsyncDroid creates an `Intent` to carry $\mathcal{OV}$, and sends it via broadcast (lines 33 to 35 and 38 to 40 in Fig. 2(b)).

Second, since `doInBackground` can invoke methods defined in `AsyncTask` or $\mathcal{TC}$'s methods, AsyncDroid copies such methods into `IntentService` class. Note that AsyncDroid moves such methods instead of copying if `doInBackground` is the only caller. However, if any of such methods are in library code and cannot be copied, the refactoring fails **P2**.

Finally, AsyncDroid rewrites the call sites of `AsyncTask.execute` into `startService`. This includes creating the `Intent` object to wrap the $\mathcal{IV}$ (lines 9-11 in Fig. 2(b)).

Notice that in Android, starting a service or sending a broadcast requests a `Context` object. AsyncDroid checks if (i) $\mathcal{TC}$ itself is a subclass of `Context` and (ii) $\mathcal{TC}$ contains a visible `Context` field or local variable, or a visible method that returns a `Context` object. AsyncDroid uses such `Context` if there is any, otherwise the refactoring stops.

Creating and Registering Receiver. `AsyncTask` communicates with GUI through handlers. However, to receive task result from `IntentService`, the developer needs to establish a channel by registering a `BroadcastReceiver` on GUI. AsyncDroid rewrites the target `AsyncTask` into `BroadcastReceiver` class. It keeps all the fields, constructors and handlers defined in `AsyncTask`, and rewrites `onPostExecute` handler into `BroadcastReceiver.onReceive` (line 17 in Fig. 2(b)). AsyncDroid also inserts statements at the beginning of onReceive to unwrap $\mathcal{OV}$ and writes them back to corresponding variables (lines 18, 19 in Fig. 2(b)).

As discussed in Sec. IV-A, to avoid losing task result during GUI destroying and recreation, the receiver should be registered in lifecycle event handlers. ASYNCDROID declares the receiver as a field of $\mathcal{TC}$ (line 3 in Fig. 2(b)). It tries to move the receiver creation and registration into $\mathcal{TC}$'s lifecycle event handlers unless they are already there. The following example shows an AsyncTask that executes in a button's onClick listener. ASYNCDROID register the receiver in onCreate lifecycle handler during refactoring instead of in the onClick listener:

```
void onCreate() {
    button.setOnClickListener() {() −> { new MyAsyncTask(...).execute();}}
}
                                        ⇓
BroadcastReceiver receiver;
void onCreate() {
  receiver = new MyBroadcastReceiver(...);
  registerReceiver(receiver, ...);
  button.setOnClickListener() {() −> {startService(...);}}
}
```

Note that a $\mathcal{TC}$ can have multiple lifecycle event handlers. ASYNCDROID registers the receiver in the handler that is invoked first by the system (e.g., onCreate). A receiver can be moved if and only if (i) the $\mathcal{TC}$ contains lifecycle event handlers, (ii) all variables transferred to the receiver are still visible to it after moving, and (iii) the variables used by receiver's constructor are not redefined in other lifecycle event handlers. Rule (ii) guarantees syntax correction while rule (iii) preserves the semantics of the refactoring. ASYNCDROID raises a warning when a receiver cannot be moved.

A filter is required when registering receiver. The filter specifies which broadcast a receiver can receive. ASYNCDROID concatenates $\mathcal{TC}$ class name and receiver name as filter name, and uses it to register receiver and send broadcast (lines 2, 8 and 29 in Fig. 2(b)).

Dealing with other AsyncTask handlers. In addition to onPostExecute, AsyncTask provides three other handlers. Note that the generated BroadcastReceiver keeps all these three handlers. For onPreExecute, ASYNCDROID inserts an invocation to this handler before starting the service.

For onProgressUpdate, ASYNCDROID first rewrites the call site of publishProgress into sendBroadcast, with the filter set to "ProgressUpdate":

```
void doInBackground(...) { publishProgress(arg); }
                                        ⇓
void onHandleIntent(...) { intent.setAction("ProgressUpdate");
     intent.putExtra("taskProgress", arg); sendBroadcast(intent); }
```

Then in onReceive, ASYNCDROID adds a branch to intercept the "ProgressUpdate" broadcast, and invokes onProgressUpdate:

```
void onProgressUpdate(...) {...}
void onReceive(Context context, Intent intent) { ... // code from onPostExecute }
                                        ⇓
void onProgressUpdate(...) {...}
void onReceive(Context context, Intent intent) {
  if(intent.getAction().equals("ProgressUpdate")) { ...; onProgressUpdate(...); }
  else ... // code from onPostExecute }
```

ASYNCDROID ignores onCancelled handler, since Android does not support canceling IntentService. Tasks that invoke AsyncTask.cancel fails **P4**.

TABLE IV: Nine popular Android projects from Github.

Project Name	SLOC	# AsyncTask	# IntentService
owncloud-android	54918	6	0
open311-android	6642	5	0
prey-android-client	15361	8	1
SMSSync	16706	10	2
opentripplanner	11766	7	0
UltimateAndroid	228154	10	0
AntennaPod	38430	24	0
WhatAndroid	23643	20	0
TextSecure	40819	18	0
Total	436439	102	3

V. EVALUATION

To empirically evaluate whether ASYNCDROID is useful, we answer the following evaluation questions.

EQ1. Applicability: How applicable is the refactoring?
EQ2. Effort: How much programmer effort is saved by ASYNCDROID when refactoring?
EQ3. Accuracy and Value: How accurate is ASYNCDROID when performing a refactoring? Do developers think that the refactorings performed by ASYNCDROID are useful?

A. Experimental Setup

To answer the above questions, we apply ASYNCDROID on nine popular Github open-source Android projects. We selected projects that use AsyncTask predominantly.

Table IV provides statistics about these projects. We report the size (in SLOC), the number of AsyncTask and IntentService instances that are used in each project.

For each project, we applied the ASYNCDROID to every AsyncTask, except for the ones that have already been used in Service or retained Fragment. The lifecycle of such AsyncTasks is independent of GUI's lifecycle, so the they do not suffer from the problems described in Sec. I.

We recorded several metrics for each refactoring. To measure the applicability, we counted how many instances met the refactoring preconditions and thus can be refactored. We also analyzed the reasons why the remaining instances cannot be refactored by ASYNCDROID. To measure refactoring effort, we recorded the number of input and output variables ($\mathcal{IV}$, $\mathcal{OV}$), serialized types, moved/copied methods and moved receivers. We also counted the number of files and SLOC that are changed. To verify the accuracy and value, we manually examine the correctness of all the refactored code. We also sent 45 refactorings in 7 projects to developers and let them judge the correctness and usefulness.

B. Results

Table V shows the result of applying ASYNCDROID on the AsyncTasks in our corpus of 9 Android projects.

Applicability. We totally refactored 97 AsyncTasks in the nine projects. Columns 2 and 4 show the number of instances that pass and fail the refactoring preconditions. There are 44 AsyncTasks that pass the preconditions, while 43 fail the preconditions. This is not a limitation of ASYNCDROID,

TABLE V: Results of applying ASYNCDROID to `AsyncTask` in nine Android projects.

Project Name	Applicability						Effort							# Task Objects
	Passed	Conditional Passed	Failed	P1	P2	P4	# $\mathcal{IV}$	# $\mathcal{OV}$	Moved Methods	Serialized Types	Moved & unmoved Receivers	Files Mod.	SLOC Mod.	
owncloud-android	1	2	3	4	1	2	6	5	1	0	2/0	3	193	3
open311-android	5	0	0	0	0	0	6	10	3	10	3/0	11	395	5
prey-android-client	7	1	0	1	0	0	11	9	5	2	4/2	10	392	8
SMSSync	1	0	6	6	0	0	0	1	0	0	1/0	1	41	1
opentripplanner	2	2	3	4	1	0	8	8	4	3	0/10	14	679	10
UltimateAndroid	5	1	4	2	2	2	6	6	0	0	4/0	7	310	8
AntennaPod	4	4	16	15	4	16	8	8	1	2	4/3	11	418	8
WhatAndroid	9	0	8	6	0	1	11	10	0	0	9/0	9	411	9
TextSecure	10	0	3	3	0	0	25	10	0	1	3/0	11	547	10
Total	44	10	43	41	7	21	81	67	14	18	30/15	77	3386	62

$\mathcal{IV}$: incoming vars; $\mathcal{OV}$: outgoing vars; **P1**: all $\mathcal{IV}$ and $\mathcal{OV}$ can be serializable; **P2**: all methods invoked by `doInBackground` are accessible in `IntentService`; **P4**: `AsyncTask` is only used when invoking `AsyncTask.execute`; **P3** is not shown because there is only one violation.

but such cases can not be converted from shared-memory to distributed-style. Column 3 shows another 10 `AsyncTasks` that fail the preconditions. However, these instances can be refactored into canonical form with other well-known refactorings (such as demoting fields to local variables) so that ASYNCDROID can refactor them. We show these instances in the "conditional pass" column.

We discovered two common transformations that convert code into canonical form. First, an unserializable object contained in $\mathcal{IV}$ but not in $\mathcal{OV}$, can be converted into a local variable in `doInBackground` as long as it is only used by `doInBackground`:

```
UnserializableObject object = new UnserializableObject(...);
void doInBackground(...) { object.method(); }
                                    ⇓
void doInBackground(...) {
    UnserializableObject object = new UnserializableObject(...); object.method(); }
```

Second, an `AscynTask` that is executed on a `ThreadPool`, can be changed to execute on a plain thread since `IntentService` does not support `ThreadPools`:

```
AsyncTask task = new AsyncTask() {...}; task.executeOnExecutor(...);
                                    ⇓
AsyncTask task = new AsyncTask() {...}; task.execute(...);
```

For refactorings that conditional-pass or fail the preconditions, we analyzed which preconditions they violate. Columns 5 to 7 shows the number of instances that fail **P1**, **P2** and **P4**. Note that one refactoring can violate multiple preconditions. The result shows most failed refactorings violate **P1**. The main reason is that the unserializable types in $\mathcal{IV}$ or $\mathcal{OV}$ are declared in third-party libraries (such as network or database). A source-to-source transformation tool like ASYNCDROID cannot transform third-party binary code.

Preconditions **P2** and **P4** are violated mainly due to methods `cancel` or `executeOnExecutor` that are invoked either in `doInBackground` (thus failing **P2**) or on the task instance (thus failing **P4**). Since these methods are specific to `AsyncTask` and `IntentService` does not support them, ASYNCDROID cannot transform those cases. For **P3**, we only find one violation in *WhatAndroid*, so we do not show them in Table V due to lack of space.

In terms of applicability, ASYNCDROID successfully refactored 45.3% `AscynTasks` directly in nine projects. There are 10.3% `AscynTasks` that can also be refactored after converting them to canonical form. This shows that ASYNCDROID has a high level of applicability.

Effort. We estimate the effort based on the 54 `AsyncTasks` that pass or conditional pass the preconditions. In the last column, we show the number of task instances that are created for the 54 `AsyncTasks`. 62 task instances are created in total, which means most `AsyncTasks` are used only at one place. This observation confirms our previous study [36]: developers tend to tightly bind an `AsyncTask` to only one GUI component.

Columns 8 and 9 show the number of $\mathcal{IV}$ and $\mathcal{OV}$ for each project. For the 54 `AsyncTasks`, there are 81 $\mathcal{IV}$ and 67 $\mathcal{OV}$. Detecting them needs inter-procedural analysis. Moreover, wrapping them into `Intent` object is also tedious.

Column 10 shows the methods that need to be moved or copied into `IntentService`. We find 14 methods that should be put into `IntentService`. Note that searching these methods also needs inter-procedural analysis. Column 11 shows the number of types that are refactored to be serializable. On average, each refactoring marks 0.33 types as serializable. However, checking serializability is tedious since it requires traversing the type hierarchy for each field.

Column 12 shows the number of `BroadcastReceivers` that are moved into lifecycle event handlers by ASYNCDROID (left side of slash), and that have to be moved but ASYNCDROID cannot move (right side of slash). Notice that for each task instance, ASYNCDROID creates a corresponding receiver. Therefore, it creates 62 receivers in total: 30 are moved, 15 cannot be moved, and the remaining 18 are already in lifecycle event handlers. For the 15 unmoved receivers, GUI component can lose task result if GUI destroying and recreation occurs during task running, thus ASYNCDROID raises a warning.

Columns 13 and 14 show the number of files and SLOC that are changed during the 54 refactorings. On average, each refactoring changes 1.43 files and 63 SLOC. ASYNCDROID helps developers change several SLOC, and such changes are non-trivial. Thus, we conclude that ASYNCDROID can save developers' effort.

Accuracy and Value. By manually examining the 54 refactored instances applied by ASYNCDROID, we determined that no compilation errors were introduced and the original semantics of `AsyncTask` are preserved.

We also submitted 45 refactorings in 7 projects to developers through Github pull request. Given the large size of changes in the patches that we submitted (on average a patch touching 10 files with 403 additions and 210 deletions), we expected that developers might not reply - as previous studies [35], [41] show that open-source developers are more likely to respond to small patches.

Despite this, by August 2015, we received replies from 4 projects in which developers accepted 15 refactorings. For example, *WhatAndroid* [18] developers accepted the 9 refactorings we submitted by saying: "*this is an interesting set of changes, AsyncTask can definitely be a pain to deal with*" and "*these tasks are a good fit for migration to IntentServices and I will migrate over to IntentServices*". *AntennaPod* [5] developers said their `AsyncTasks` are short: "*most of our tasks read or write data from the database and should finish well under 100ms*". They think `AsyncTasks` work fine for their short-running tasks while `IntentService` makes the code more verbose. This shows that ASYNCDROID can produce accurate and valuable results.

VI. RELATED WORK

Performance analysis and testing for mobile apps. Liu et al. [37] empirically study performance bug patterns in Android apps, and conclude that executing long-running operations in main thread is the main culprit. Berardinelli et al. [24] introduce a framework for modeling and analyzing the performance of context-aware mobile software systems. Arijo et al. [19] propose a model-based approach to analyze the performance of mobile apps, in which they represent state changes by graph transformation. Muccini et al. [39] analyze the challenges in testing mobile apps, including performance and memory testing. Lillack et al. [34] propose an approach to track load-time configuration for Android apps, which can help with tuning performance of Android apps. Yan et al. [49] propose a test generation approach to detect memory leaks for Android apps. However, our work is complementary to previous approaches: we assume that developers have already used the above techniques to detect responsiveness problems in their apps, and now we enable developers to fix these problems via refactoring for async programming.

Empirical study on concurrency, refactoring and API usage. Li et al. [33] study and categorize bug characteristics in modern software. Their result shows concurrency and performance related bugs can have a severe impact on software. Bavota et al. [23] investigate to what extent refactoring activities induce faults. There are also several empirical studies [26], [28], [29], [31], [43] on the usage of libraries or programing language constructs. Sahin et al. [44] analyzed how refactorings affect energy usage. Buse et al. [25] propose an automatic technique for synthesizing API usage examples and conduct a study on the generated examples. Our previous work [35], [41] studies the misuses of concurrent constructs in Java and C#. Our formative study on usage of async constructs is similar with other studies in the literature, but on different topics. In this work, we study how developers use Android async constructs.

Refactoring for concurrency, parallelism and asynchrony. The refactoring community has been recently pushing refactoring technology beyond its classic realm (i.e. in improving software design) into improving non-functional qualities such as performance through parallelism and concurrency. Schafer et al. [46] propose a refactoring for replacing Java built-in locks with more flexible locks. Wloka et al. [48] present a refactoring for replacing global state with thread local state. Schafer et al. [45] examine whether classic refactorings can be safely applied to concurrent programs. Our group implemented several implemented several concurrency-related refactorings to improve *throughput* [27], [32], [47]. Our previous work [36] presents a refactoring for Android developers to extract synchronous code from UI thread into async code that inverts the flow of control. In this work, we go one step further and investigate a new scientific challenge: converting between shared-memory communication with the UI into a distributed-style as encompassed in two Android async constructs.

Compiling shared-memory to distributed-memory. Several compiler techniques [20]–[22], [38], [51] have attempted to translate shared-memory program to distributed-memory program. However, these techniques target high performance distributed computing. In our work, we presented a refactoring from a shared-memory construct to a distributed-style construct in the context of Android asynchrony.

VII. CONCLUSIONS

Asynchronous execution of long-running tasks is crucial for the now ubiquitous mobile and wearable apps. Despite significant efforts to educate Android app developers on how to use async programming, developers can improperly use the primary construct, `AsyncTask`, which can lead to memory leaks, lost results, and wasted energy.

In this paper we take stake of the usage of async constructs in a corpus of 611 widely-used Android apps. We discovered that developers refactor their sync code into `AsyncTask`, and some go further into using safer (but more complex) async constructs. To aid developers when converting to these constructs, we designed, implemented, and evaluated ASYNCDROID. It is a refactoring that converts from `AsyncTask` which uses a shared-memory style of communication to `IntentService` which uses a distributed style of communication. Our empirical evaluation shows that ASYNCDROID is applicable and accurate, and it saves effort. Developers already accepted several refactorings generated by ASYNCDROID, which shows that it is valuable.

VIII. ACKNOWLEDGMENTS

We would like to thank Michael Hilton, Mihai Codoban, Kendall Bailey, Shane McKee and Sruti Srinivasa for their feedback on earlier versions of this paper. This research is partly funded through NSF CCF-1439957, CCF-1442157 grants, a SEIF award from Microsoft, and a gift grant from Intel.

References

[1] "Activitys, Threads and Memory Leaks," May 2015, http://www.androiddesignpatterns.com/2013/04/activitys-threads-memory-leaks.html.

[2] "Android Intents and Intent Filters," May 2015, http://developer.android.com/guide/components/intents-filters.html.

[3] "Android Processes and Threads," May 2015, http://developer.android.com/guide/components/processes-and-threads.html.

[4] "Android's AsyncTask," May 2015, http://steveliles.github.io/android_s_asynctask.html.

[5] "Antennapod repository." May 2015, https://github.com/AntennaPod/AntennaPod.

[6] "The dark side of AsyncTask," May 2015, http://bon-app-etit.blogspot.com/2013/04/the-dark-side-of-asynctask.html.

[7] "Eclips Refactoring Engine," May 2015, https://www.eclipse.org/articles/Article-LTK/ltk.html.

[8] "Eclipse Java development tools (JDT)," May 2015, http://www.eclipse.org/jdt/.

[9] "Gartner." May 2015, http://www.gartner.com/newsroom/id/2153215.

[10] "GiTective," May 2015, https://github.com/kevinsawicki/gitective.

[11] "GitHub," May 2015, https://github.com.

[12] "Java Serializability," May 2015, http://docs.oracle.com/javase/1.5.0/docs/guide/serialization/spec/serial-arch.html.

[13] "JDK Swing Framework," May 2015, http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[14] "SLOCCount," May 2015, http://www.dwheeler.com/sloccount/.

[15] "Stack Overflow." May 2015, http://stackoverflow.com.

[16] "Tablet Sales." May 2015, http://www.gartner.com/newsroom/id/2954317.

[17] "The SWT Toolkit," May 2015, http://eclipse.org/swt/.

[18] "Whatandroid repository." May 2015, https://github.com/Gwindow/WhatAndroid.

[19] N. Arijo, R. Heckel, M. Tribastone, and S. Gilmore, "Modular performance modelling for mobile applications," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '11, 2011, pp. 329–334.

[20] P. Banerjee, J. Chandy, M. Gupta, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su, "The paradigm compiler for distributed-memory message passing multicomputers," *IEEE Computer*, vol. 28, pp. 37–47, 1994.

[21] A. Basumallik and R. Eigenmann, "Towards automatic translation of openmp to mpi," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05, 2005, pp. 189–198.

[22] ——, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06, 2006, pp. 119–128.

[23] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12, 2012, pp. 104–113.

[24] L. Berardinelli, V. Cortellessa, and A. D. Marco, "Performance modeling and analysis of context-aware mobile software systems," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, ser. FASE 10, 2010, pp. 353–367.

[25] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 782–792.

[26] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger, "How developers use the dynamic features of programming languages: The case of Smalltalk," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 23–32.

[27] D. Dig, J. Marrero, and M. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 397–407.

[28] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of Java language features," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 779–790.

[29] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, 2010, pp. 11:1–11:10.

[30] W. G. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, University of Washington, Seattle, WA, USA, 1992.

[31] S. Karus and H. Gall, "A study of language usage evolution in open source software," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 13–22.

[32] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for class immutability," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 61–70.

[33] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: An empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ser. ASID '06, 2006, pp. 25–33.

[34] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014, pp. 445–456.

[35] Y. Lin and D. Dig, "Check-then-act misuse of Java concurrent collections," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '13, 2013, pp. 164–173.

[36] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 341–352.

[37] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 1013–1024.

[38] S.-J. Min and R. Eigenmann, "Optimizing irregular shared-memory applications for clusters," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008, pp. 256–265.

[39] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *Proceedings of the 7th International Workshop on Automation of Software Test*, ser. AST '12, 2012, pp. 29–35.

[40] M. Murphy, *The Busy Coder's Guide to Android Development*. CommonsWare, 2009.

[41] S. Okur, D. Hartveld, D. Dig, and A. Deursen, "A study and toolkit for asynchronous programming in C#," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 1117–1127.

[42] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[43] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of Java generics," *Empirical Softw. Engg.*, vol. 18, no. 6, pp. 1047–1089, Dec. 2013.

[44] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14, 2014, pp. 36:1–36:10.

[45] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent Java code," in *Proceedings of the 24th European Conference on Object-oriented Programming*, ser. ECOOP'10, 2010, pp. 225–249.

[46] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Refactoring Java programs for flexible locking," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 71–80.

[47] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson, "Inferring method effect summaries for nested heap regions," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, 2009, pp. 421–432.

[48] J. Wloka, M. Sridharan, and F. Tip, “Refactoring for reentrancy,” in *Proceedings of the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. FSE ’09, 2009, pp. 173–182.

[49] D. Yan, S. Yang, and A. Rountev, “Systematic testing for resource leaks in Android applications,” in *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering*, ser. ISSRE 13, 2013, pp. 411–420.

[50] S. Yang, D. Yan, and A. Rountev, “Testing for poor responsiveness in Android applications,” in *Proceedings of the International Workshop on the Engineering of Mobile-Enabled Systems*, ser. MOBS ’13, 2013, pp. 1–6.

[51] J. Zhu, J. Hoeflinger, and D. Padua, “Compiling for a hybrid programming model using the lmad representation,” in *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC ’01, 2003, pp. 321–335.