Retrofitting Concurrency for Android Applications through Refactoring

Yu Lin
University of Illinois
yulin2@illinois.edu

Cosmin Radoi
University of Illinois
cos@illinois.edu

Danny Dig
Oregon State University
digd@eecs.oregonstate.edu

ABSTRACT

Running compute-intensive or blocking I/O operations in the UI event thread of smartphone apps can severely degrade responsiveness. Despite the fact that Android supports writing concurrent code via `AsyncTask`, we know little about how developers use `AsyncTask` to improve responsiveness. To understand how `AsyncTask` is used/underused/misused in practice, we first conduct a formative study using a corpus of 104 open-source Android apps comprising 1.34M SLOC. Our study shows that even though half of the apps use `AsyncTask`, there are hundreds of places where they missed opportunities to encapsulate long-running operations in `AsyncTask`. Second, 46% of the usages are manually refactored. However, the refactored code contains concurrency bugs (such as data races) and performance bugs (concurrent code still executes sequentially).

Inspired by these findings, we designed, developed, and evaluated Asynchronizer, an automated refactoring tool that enables developers to extract long-running operations into `AsyncTask`. Asynchronizer uses a points-to static analysis to determine the safety of the transformation. Our empirical evaluation shows that Asynchronizer is (i) highly applicable, (ii) accurate, (iii) safer than manual refactoring (iv) it saves development effort, (v) its results have been accepted by the open-source developers. This shows that Asynchronizer is useful.

1. INTRODUCTION

For smartphone apps, responsiveness is critical. The apps can easily be unresponsive because mobile devices have limited resources and have high latency (excessive network accesses). With the immediacy of touch-based UIs, even small hiccups in responsiveness are more obvious and jarring than when using a mouse or keyboard. Some sluggishness might motivate the user to uninstall the app, and possibly submit negative comments in the app store.

Previous research [27, 37] shows that many Android apps suffer from poor responsiveness and one of the primary reasons is that apps execute all workload in the UI event thread. The UI event thread of an Android app processes UI events, but long-running operations (i.e., CPU-bound or blocking I/O operations) will "freeze" it, so that it cannot respond to new user input or redraw. Android documentation [1] warns that developers should not use long-running operations in the UI event thread.

The primary way to avoid freezing the UI event thread is to resort to concurrency, by extracting long-running operations into a background thread. While programmers can use `java.lang.Thread` to fork concurrent asynchronous execution, it is cumbersome to communicate with the main thread. Android framework provides a better alternative, `AsyncTask`, which is a high-level concurrent construct. `AsyncTask` can also interact with the UI thread by updating the UI via event handlers. For example, the event handler `onPostExecute` executes after the task is finished, and can update the UI with the task results.

In this paper we first present a formative study to understand how developers use `AsyncTask`. We analyzed a corpus of 104 popular, open-source Android apps, comprising 1.34M SLOC, produced by 1139 developers. The formative study answers the following questions:

RQ1: How is `AsyncTask` used? We found that 48% of the studied projects use `AsyncTask` in 231 different places. In 46% of the uses, developers extracted long-running operations into `AsyncTask` via manual refactoring. In the remaining cases, they used `AsyncTask` from the first version.

RQ2: How is `AsyncTask` misused? We found two kinds of misuses. First, in 4% of the invoked `AsyncTask`, the code runs sequentially instead of concurrently: the code launches an `AsyncTask` and immediately blocks to wait for the task's result. We found similar problems in our previous studies on concurrent libraries in C# [28, 29]. Second, we found that in 13 cases, code in `AsyncTask` accesses GUI widgets in a way which is not thread-safe. This leads to data races on these GUI widgets.

RQ3: Is `AsyncTask` underused? We found that 251 places in 51 projects execute long-running operations in UI event thread. This also confirms the findings of a recent study by Liu et al. [27] that shows that 21% of reported responsiveness bugs in an Android corpus arise because developers tend to forget encapsulating long-running operations in `AsyncTask`.

Inspired by these findings, we designed, developed, and evaluated Asynchronizer, an automated refactoring tool that enables developers to use `AsyncTask`. To perform the refactoring, the developers only need to select the code that they want to run in background. ASYNCHRONIZER will automatically create an instance of `AsyncTask` as an inner class, generate event handlers in `AsyncTask`, and start the task.

However, manually applying this refactoring is non-trivial. First, a developer needs to reason about fields, method arguments, and return value for `AsyncTask`, and the statements that can be moved into `AsyncTask`. This requires reasoning about control- and data-flow. Second, the developer has to deal with several special cases. For example, `this` or `super` are relative to the enclosing class where they are used, whereas after extracting them into `AsyncTask`, they are relative to the inner `AsyncTask` class. Third, the developer needs to analyze the read-write effects on shared variables in order to prevent introducing data races.

To solve these challenges, we decompose the refactoring into two steps: code transformation and safety analysis. The transformation part uses Eclipse's refactoring engine to rewrite the code. The safety analysis uses a static race detection approach, specialized to `AsyncTask`. We implemented it as an extension of the ITERACE [30] race detector.

This paper makes the following contributions:

1. Formative study: To the best of our knowledge, this paper presents the first study on the usage, misusage, and underusage of `AsyncTask` in Android apps.

2. Algorithms: We designed the analysis and transformation algorithms to address the challenges of refactoring long-running operations into `AsyncTask`. The algorithms account for inversion of control by transforming sequential code into callback-based asynchronous code, and reason about non-interference of updates on shared variables.

3. Tool: We implemented the refactoring in a tool, ASYNCHRONIZER, which integrates with the Eclipse refactoring engine.

4. Tool Evaluation: To evaluate ASYNCHRONIZER's usefulness, we used it to refactor 135 places in 19 open-source Android projects. We evaluate ASYNCHRONIZER from five angles. First, since 95% of the cases meet refactoring preconditions, it means that the refactoring is highly applicable. Second, in 99% of the cases, the changes applied by ASYNCHRONIZER are similar with the changes applied manually by open-source developers, thus our transformation is accurate. Third, ASYNCHRONIZER changes 2394 LOC in 62 files in just a few seconds per refactoring. Fourth, using ASYNCHRONIZER we discovered and reported 169 data races in 10 apps. 5 replied and confirmed 62 races. This shows that the automated refactoring is safer than manual refactoring. Fifth, we also submitted patches for 58 refactorings in 6 apps. 4 replied and accepted 10 refactorings. This shows that ASYNCHRONIZER is valuable.

The tool and experimental subjects are available at:
`http://mir.cs.illinois.edu/~yulin2/asynchronizer`

2. BACKGROUND ON ANDROID PROGRAMMING

2.1 Android GUI Programming

Android GUIs are typically composed of several *activities*. An activity represents a single GUI screen. For example, the login screen of an email client is an activity. An application GUI transitions through a sequence of activities, each of which is independent of the others. However, at any given time, only one activity is active. Activities contain GUI widgets.

```
public class MainActivity extends Activity {
    public boolean onOptionsItemSelected(MenuItem item) {
        ...
        new Button(new OnClickListener() {
            public void onClick(...) {
                exportToSpreadsheet(gameIds);
            }
        });
    }
    private void exportToSpreadsheet(final List gameIds) {
        ...
        new AsyncTask<Void, Void, String>(){
            protected void onPreExecute(String filename) { }
            protected String doInBackground(Void... params) {
                ...
                for (Object gameId : gameIds) {
                    games.add(dbHelper.findGameById(gameId));
                    publishProgress((Void)null);
                }
                String filename = ...
                return filename;
            }
            protected void onProgressUpdate(Void... values) {
                progressDialog.incrementProgressBy(1);
            }
            protected void onPostExecute(String filename) {
                progressDialog.dismiss();
            }
            protected void onCancelled (String filename) { }
        }.execute((Void)null);
    }
}
```

Figure 1: Real-world example of `AsyncTask` in `KeepScore`

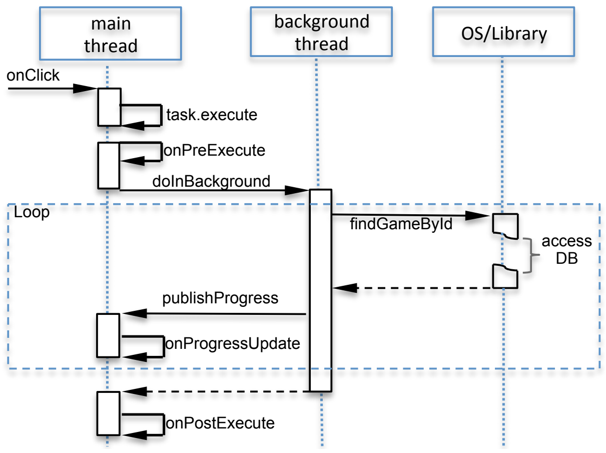

Figure 2: Where is `AsyncTask` code executing?

Similar to many other GUI frameworks such as Swing [7] and SWT [9], Android uses an event-driven model. Events in Android apps 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 invoked when an activity is created, while `onClick` handler of a button is invoked when a button is clicked.

Android framework uses a **single thread model** to process events [1]. When an application is launched, the system creates a *main thread*, i.e., the *UI event 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. The main thread puts events into a single event queue, dequeues events, and executes corresponding event handlers.

However, if the main thread executes CPU-intensive work or blocking I/O calls such as network access or database queries, this results in poor responsiveness. Once the main thread is blocked, no events can be dispatched and processed, so application will not be responsive to users' actions. To avoid unresponsiveness, developers should exploit concurrency and extract long-running operations into another thread.

2.2 `AsyncTask` in Android Framework

To ease the use of concurrency, Android framework provides `AsyncTask` class. `AsyncTask` is a high-level abstraction for encapsulating concurrent work. `AsyncTask` also provides event handlers such as `onPostExecute` that execute on the main thread after the task has finished. Thus, the background task can communicate with the main thread via these event handlers.

We illustrate a typical usage of `AsyncTask` using a real-world app, `KeepScore`, that keeps scores for games that require tallying points as one plays. Figure 1 shows an `AsyncTask` that reads game scores from a database and exports them in a spreadsheet file. The methods that start with "`on`" are event handlers. Figure 2 shows the flow of this `AsyncTask`.

Line 4 sets up a listener for a button, and when the button is clicked, method `exportToSpreadsheet` is called. This method creates an `AsyncTask` (line 12) and executes it concurrently with the main thread. `doInBackground` method (line 14) encapsulates the work that executes in the background. The task queries database and adds the results to a list, `games` (line 17). Finally, the result of the background computation is returned (i.e., `filename` at line 21).

While the task is executing, it can report its progress to the main thread by invoking `publishProgress` and implementing `onProgressUpdate` handler. In the example, the task publishes its progress every time it finds a game (line 18), so the main thread can update a progress dialog (line 24). The main thread executes the `onPostExecute` handler after `doInBackground` finishes. In this example, the handler updates the GUI by dismissing the progress dialog (line 27). Notice that this handler takes the result of the task as parameter (`filename` at line 26).

When it manages the lifecycle of an `AsyncTask`, the main thread executes `onPreExecute` (line 13) before the `doInBackground`. It also executes the `onCancelled` (line 29) when the task is cancelled.

The three generic types of `AsyncTask` (line 12) represent the parameter types of `doInBackground`, `onProgressUpdate`, and the return type of `doInBackground`.

Notice that there are two ways that the main thread can fetch the result of a `AsyncTask`. One, the result is available in the `onPostExecute`. Second, the result can be explicitly requested through the `get` method on the task. This method has blocking semantics: if the result is available, it will return immediately, otherwise it will block the main thread until the result becomes available.

3. FORMATIVE STUDY ON USAGE OF `ASYNCTASK`

In this section we present our formative study to understand how developers use, misuse, and underuse `AsyncTask` in open-source Android apps.

3.1 Experimental Setup

Corpus of Android Apps.

We selected our corpus of Android apps from Github [6]. To find Android apps, we filter Java repositories by searching if their `README` file contains "Android app" keyword. We also manually confirmed that these repositories are Android apps. We apply two more filters. First, because we want to contact developers, we need to avoid analyzing inactive projects. Thus, we only keep repositories that contain at least one commit after June 2013. Second, because we want to study the usage of `AsyncTask` in mature, representative apps, we ignore "toy applications", i.e., apps that have less than 500 SLOC. Also, we ignore all forked applications since they are similar to the original repository. Finally, we use the top 104 most popular projects as our corpus, comprising 1.34M SLOC, produced by 1139 developers.

Analysis.

We want to study whether developers refactor existing code into `AsyncTask` (i.e., they encapsulate existing statements into `AsyncTask`), or whether they introduce `AsyncTask` on new code they write from scratch. Thus, we study not the latest version of the code which contains `AsyncTask`, but the first version of the code where developers introduce this construct. To do this, we searched the commit history of our corpus through GiTective API [5], identified the commits that add import statements to `AsyncTask`, and manually inspected the versions before and after such commits. This helps us understand questions about correct and incorrect usage.

To understand whether the corpus contains underusage of `AsyncTask`, we want to identify long-running operations that execute in the UI event thread and are potentially decreasing the responsiveness of the application. These operations are candidates to be encapsulated within `AsyncTask`.

Thus we first created a "black list" of long-running operations that Android documentation [1] warns should be avoided in the UI. We used Eclipse's search engine to find call sites to such operations. Using the call graph hierarchy, we analyzed whether they appear directly or indirectly in event handlers but not in `AsyncTask` or `Thread`.

To assure validity and reliability, we make all the data-set and the results available online [2].

3.2 Results

Table 1 shows the results about usage and misusage in the 50 projects that use `AsyncTask`. The second row shows the items we count, including the number of instances of `AsyncTask` (column 2), how many event handlers of `AsyncTask` are implemented by developers (columns 3 to 6). number of misuse which includes accessing GUI in `doInBackground` (column 7) and wrong usage of `get` (column 8). The third row counts these items in newly introduced `AsyncTask` (i.e., code where developers use `AsyncTask` from scratch). The forth row counts these items in code that was manually refactored by developers to use `AsyncTask`. The fifth row sums the total usage in newly introduced and refactored `AsyncTask`.

Table 1: AsyncTask usage and misuage.

	Usage					Misusage	
Item	AsyncTask instances	onPost-Execute	onPre-Execute	onProgre-ssUpdate	onCancelled	GUI access	Wrong usage of get
Newly introduced	125	123	44	7	6	6	0
Manually Refactored	106	64	10	2	0	7	9
Total	231	187	54	9	6	13	9

Using the data in Tab. 1, we answer the following three questions:

RQ1: How is `AsyncTask` used?
50 out of 104 projects use `AsyncTask` to embrace concurrency. This shows `AsyncTask` is adopted in Android apps.

54% (125 out of 231) of `AsyncTask` instances are newly introduced when developers add new features. However, there are 46% (106 out of 231) `AsyncTask` refactored. Here we found two refactoring scenarios. First, in 94 cases, the sequential code was refactored directly into `AsyncTask`. Second, in 12 cases, the code is refactored from regular Java `Thread` into `AsyncTask`. This is reasonable since `AsyncTask` provides event handlers and is easier to use than `Thread` when the background thread needs to communicate with the main thread.

Lastly, we notice that `onPostExecute` handler is the most widely implemented by developers (81% (187 out of 231)). However, for the other three handlers, the implementation percentage is only 23%, 4% and 3%. A possible explanation is that after a task is done, applications have to update UI and report the result to users. `onPostExecute` handler provides an easy way to update UI without explicitly knowing when the task is finished, so it is implemented in most cases.

RQ2: How is `AsyncTask` misused?
We found that 13 `AsyncTask` (7 in manual refactoring) access GUI in `doInBackground`. However, based on the Android document, accessing GUI from outside main thread will lead to races, because Android UI toolkit is not thread-safe.

Data races can also occur on the non-GUI objects after developers transform sequential code to concurrent code. Figure 3 shows a manual refactored `AsyncTask` in `ChatSecureAndroid` project. At line 3, `onLoadFinished` handler eventually calls `startGroupChat` method, in which an `AsyncTask` is executed. This task writes to field `mRequestedChatId` at line 17. However, this field is read at line 4, which can be executed concurrently with line 17. Thus, there is a race on `mRequestedChatId`. Note that this data race is found by Asynchronizer in our evaluation (see Sec. 6) rather than being found manually in this formative study.

Also, nine manually refactored `AsyncTask`s are misused because developers invoke `get` method on the task immediately after starting the task. As we mentioned in Sec. 2, invocation of `get` blocks the current thread until the result is available. Thus, such usage blocks the main thread immediately and defies the purpose of using `AsyncTask`.

```
1   public class NewChatActivity extends SherlockFragmentActivity {
2       public void onLoadFinished(Loader loader, Cursor newCursor) {
3           resolveIntent();
4           if (mRequestedChatId >= 0) {
5               ...
6           }
7       }
8       private void resolveIntent() {
9           startGroupChat(path, host, listConns.get(0));
10          ...
11      }
12      private void startGroupChat(...) {
13          ...
14          new AsyncTask<String, Void, String>() {
15              protected String doInBackground(String... params) {
16                  ...
17                  mRequestedChatId = session.getId();
18              }
19          }.execute(room, server);
20      }
21  }
```

Figure 3: In `ChatSecureAndroid` project, developers introduce races in manual refactoring.

RQ3: Is `AsyncTask` underused?
We found that 51 out of 104 projects call long-running APIs in UI event handlers at 251 places. In these 51 projects, 17 projects have already used `AsyncTask`. Still, we found 79 places where `AsyncTask` is underused. The remaining 34 projects never use `AsyncTask`.

Based on our findings for RQ1, we conclude that `AsyncTask` is widely adopted and developers manually refactor their code to use `AsyncTask` in many cases. However, as RQ3 shows, `AsyncTask` is still underused. RQ2 shows that manual refactoring may introduce bugs.

Based on the results for RQ1–RQ3, we conclude that there is a need for safe refactoring tools to enable developers to transform code towards `AsyncTask` (presented in Sec. 4), as well as help developers check possible races that can occur between the code running in `AsyncTask` and the code running in the main thread (presented in Sec. 5).

4. TRANSFORMATIONS

This section describes the code transformation that enables developers to move code from main thread into `AsyncTask`. We implement the transformation in our tool, Asynchronizer. We first explain the overall workflow of the tool, and then illustrate the transformations.

4.1 Refactoring Workflow and Preconditions

We implement Asynchronizer as a plugin in the Eclipse IDE [3]. To use Asynchronizer, the programmer selects statements that she wants to encapsulate within `AsyncTask`, and then chooses the Convert to AsyncTask option from the refactoring menu. The programmer can also specify the class and instance name that Asynchronizer will use to generate `AsyncTask`. Asynchronizer moves the selected statements into `AsyncTask.doInBackground` method. In addition, Asynchronizer also infers the subsequent statements that can be moved to `onPostExecute`. Before applying the changes, Asynchronizer gives the programmer the option to preview them in a before-and-after pane. Then, Asynchronizer transforms the code in place.

After the transformation, the programmer can invoke Asynchronizer's safety analysis component to check data

```
1  public class RouteselectActivity extends ListActivity {
2      ...
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          ...
6          ListView lv = getListView();
7          final String qry = "select...";
8          final String[] selectargs = {mStopid, datenow, datenow};
9
10
11
12
13
14
15
16
17
18
19
20         Cursor mCsr = DatabaseHelper.ReadableDB()
21             .rawQuery(qry, selectargs);
22         startManagingCursor(mCsr);
23
24
25
26         lv.setOnTouchListener(mGestureListener);
27         if (mCsr.getCount() > 1)
28             tv.setText(R.string.route_fling);
29         else if (mCsr.getCount() == 0)
30             tv.setText(R.string.stop_unused);
31         lv.addFooterView(tv);
32         CursorAdapter adapter =
33             new CursorAdapter(this, mCsr);
34         setListAdapter(adapter);
35     }
36
37 }
```

(a) before

```
1  public class RouteselectActivity extends ListActivity {
2      ...
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          ...
6          ListView lv = getListView();
7          final String qry = "select...";
8          final String[] selectargs = {mStopid, datenow, datenow};
9          ProcessRoutes prTask = new ProcessRoutes(lv);
10         prTask.execute(qry, selectargs);
11     }
12     class ProcessRoutes extends AsyncTask<Object, Void, Cursor>{
13         ListView lv;
14         ProcessRoutes(ListView lv) {
15             this.lv = lv;
16         }
17         protected Cursor doInBackground(Object... args) {
18             String qry = (String) args[0];
19             String[] selectargs = (String[]) args[1];
20             Cursor mCsr = DatabaseHelper.ReadableDB()
21                 .rawQuery(qry, selectargs);
22             startManagingCursor(mCsr);
23             return mCsr;
24         }
25         protected void onPostExecute(Cursor mCsr) {
26             lv.setOnTouchListener(mGestureListener);
27             if (mCsr.getCount()>1)
28                 tv.setText(R.string.route_fling);
29             else if (mCsr.getCount() == 0)
30                 tv.setText(R.string.stop_unused);
31             lv.addFooterView(tv);
32             CursorAdapter adapter =
33                 new CursorAdapter(RouteselectActivity.this, mCsr);
34             setListAdapter(adapter);
35         }
36     }
37 }
```

(b) after

Figure 4: Relevant code from GR-Transit app. Programmer selects lines 20 to 22 in (a), and Asynchronizer performs all the transformations. The left-hand side shows the original code, whereas the right-hand side shows the refactored code by Asynchronizer.

races due to the transformation. We will present the safety analysis in Sec. 5. If Asynchronizer found data races, the programmer still needs to confirm and fix them manually. Only after this the refactored code is correct.

Figure 4(a) shows a code snippet from an Android app, `GR-Transit`, that displays bus routes and schedules . The code snippet is used to show the bus routes that pass a given bus stop. If the programmer applies our transformation to lines 20 to 22, Asynchronizer will transform the code to Fig. 4(b). In a subsequent version of `GR-Transit`, the programmers have done this transformation manually. Their new code, modulo syntactic difference, is the same as Asynchronizer's output.

Asynchronizer checks the following three preconditions before transforming, and reports failed preconditions.

(P1) The selected statements do not write to more than one variable which is read in the statements after the selection. Such a variable needs to be returned by `doInBackground`, but Java methods can only return one single variable.

(P2) The selected statements should not contain `return` statements. A `return` statement in the original code enforces an exit point from the enclosing method. However, the same `return` statement extracted into an `AsyncTask` can no longer stop the execution of the original method.

(P3) The selection contains only entire statements. Selecting an expression that is part of a statement is not allowed because it would force the `AsyncTask` to immediately invoke the blocking `AsyncTask.get()` to fetch the expression; this defies the whole purpose of launching an `AsyncTask`.

4.2 Create `doInBackground` Method

The first step of the transformation is to move the selected statements into `doInBackground` method. This is similar to Extract Method refactoring, in which programmer moves a piece of code into a new method. In this step, Asynchronizer need to determine the arguments and the return value of `doInBackground`. The arguments are the local variables which are used in the selection but declared before it. The return value is the local variable which is defined in the selection but used after it.

In Fig. 4(b), the `doInBackground` method takes two arguments, `qry` and `selectargs`, and returns `mCsr`. However, note that `doInBackground` has only one *varargs* parameter (i.e., array of unknown length), and its type is specified by the type argument (generic) of `AsyncTask`. If all local variables are of the same type, Asynchronizer sets this type as the first generic type argument for `AsyncTask`. If the passed-in local variables are of different types, as it is the case for our example, Asynchronizer uses `java.lang.Object` as the generic type argument (Fig. 4(b) line 17), and dereferences and type-casts the parameters (Fig. 4(b) lines 18 and 19). If `doInBackground` has no arguments or return value, it uses `Void` as parameter type or return type, and returns `null`.

4.3 Create `onPostExecute` Handler

The second step is to infer which code can be put into `onPostExecute` handler. Because the Android framework invokes the `onPostExecute` after the method `doInBackground` has finished, the analysis needs to determine that the statements inside these two methods follow the same control-flow as in the original program.

A naive implementation is to move all the statements after the selected code into `onPostExecute`. However, this may break the control flow of the main thread. A statement cannot be moved if it is not dominated by the statements in the selected code, or if it is a `return` statement. A statement *dominates* [11] another if every path from the entry point to the latter statement passes through the former statement.

Algorithm 1 infers the set of statements to be moved to `onPostExecute`. The inputs of the algorithm include the selected code that will be put into `doInBackground` (*selected*), the list of statements syntactically after the selected code (*post*), and the return variable of `doInBackground` (*rv*). The output is the set of statements which can be moved to `onPostExecute` (*moved*). *unmoved* contains the statements which cannot be moved.

The algorithm first selects the prefix of *post* in which all statements are dominated by *selected* and do not `return` (lines 3 to 9). The remaining statements cannot be moved so they are put into the *unmoved* variable (line 10). The algorithm then constructs the final result as the prefix of *dominated* for which all statements have no effect on any statement in *dominated* (lines 11 to 18). This ensures no data dependencies are broken. *unmoved* is updated with any statements which are not in *moved* (line 19). Finally, if *unmoved* contains statements that use the resulting value of `doInBackground`, Asynchronizer adds a call to `AsyncTask`'s `get` method before the first such use (lines 20 to 22).

In the example shown in Fig. 4(a), all the statements after the selected code (lines 26 to 34) can be put into `onPostExecute`. However, suppose there was a statement at line 35 that returns `mCsr`. Such a statement would not be moved to `onPostExecute`. Furthermore, Asynchronizer would add a call to `AsyncTask.get` before this return statement because it uses `mCsr`. In the current implementation, Asynchronizer uses Eclipse JDT's [4] variable bindings to approximate data dependencies.

4.4 Create Class Declaration

In this step, Asynchronizer creates fields, constructor and class declaration for `AsyncTask`. Fields are generated by analyzing the statements in `onPostExecute`. Since `onPostExecute` only have one parameter which is the return value of `doInBackground`, the tool converts all the other arguments needed by `onPostExecute` into fields of `AsyncTask`. For example, in Fig. 4(b), local variable `lv` is needed by `onPostExecute`. Asynchronizer declares a field `lv` in the `AsyncTask` (line 13) and adds a constructor to initialize this field (line 14). After that, it creates an inner class declaration using all the code elements which have been created above (line 12). Finally, it generates two statements to create task instance and call `execute` method, and replaces the selected code by these two statements (lines 9 and 10).

4.5 Special Cases

Asynchronizer also analyzes code to properly transform several special cases:

Algorithm 1 $inferringPostStmts(selected, post, rv)$

```
Input: selected ← the selected code
       post ← all statements after the selected code
       rv ← return variable of doInBackground
Output: moved ← statements put into onPostExecute
 1: dominated ← []
 2: unmoved ← []
 3: for all stmt in post do
 4:     if selected dominates stmt and not stmt contains return then
 5:         dominated ← dominated append stmt
 6:     else
 7:         break
 8:     end if
 9: end for
10: unmoved ← post − dominated
11: moved ← []
12: for all stmt in dominated do
13:     if not unmoved is data dependent on stmt then
14:         moved ← moved append stmt
15:     else
16:         break
17:     end if
18: end for
19: unmoved ← post − moved
20: if unmoved uses rv then
21:     invoke get method before the first use of rv in unmoved
22: end if
23: return moved
```

(S1) `doInBackground` and `onPostExecute` cannot be declared to throw checked exceptions. Thus, if the selected statements throw exceptions (e.g., programmer selects `FileOutputStream.write` method which throws `IOException`), Asynchronizer needs to generate try-catch block to handle the exceptions. Asynchronizer first collects the exceptions that are declared to be thrown by the selected code. If these exceptions are caught in the original refactored method, it copies the corresponding catch clauses into `doInBackground` or `onPostExecute` to handle the exceptions. Otherwise, it generates empty catch clause. In our experiment, all the cases that throw exceptions have corresponding catch clauses in the original code.

(S2) The original code may use `this` or `super` pointer (Fig. 4(a) line 33). After moving it to an inner `AsyncTask` class, our tool replaces the original pointer with outer class' `this` or `super` pointer (Fig. 4(b) line 33).

5. DATA RACE CHECK

Our formative study (Sec. 3) shows that developers do introduce data races when they manually refactor sequential code into `AsyncTask` concurrent code. These data races are either accesses to GUI elements from the `doInBackground`, or possibly concurrent accesses to other shared resources. Data races are hard to find as they only manifest themselves under certain thread schedules. To assist developers with the refactoring, we propose a static race detection approach specialized to the thread structure generated by `AsyncTask`. We implement our approach as an extension of the IteRace race detector [30].

IteRace is a static race detector for Java parallel loops that achieves low rates of false warnings by taking advantage of the extra semantic information provided by the use of high-level concurrency constructs. It uses the known thread-safety properties of concurrent collection classes, summarizes races that occur in libraries at the level of the application, and specializes for the thread structure of lambda-style parallel loops.

While ITERACE is only capable of analyzing parallel loops, its approach of taking advantage of the implicit thread structure of high-level concurrency constructs is also applicable to `AsyncTask`. We thus extend ITERACE to find races that occur between `doInBackground` and other threads.

5.1 Data races

Generally, a data race is a pair of accesses, one of which is a write, to the same memory location, with no ordering constraint between the two accesses. For `AsyncTask`, a data race can occur between accesses in `doInBackground` and accesses which may execute in parallel with the asynchronous task. While the precise set of instructions that may execute in parallel cannot be determined statically, we can find an approximation of it.

ASYNCHRONIZER relies on the Andersen-style static pointer analysis [13] provided by WALA [10]. Thus, our analysis works over an abstract heap built along with a (k-bounded) context-sensitive call graph. The underlying analysis is flow insensitive, except for the limited sensitivity obtained from the SSA form.

Our tool makes the following approximation for a race:

Instruction i_α in call graph node n_α races with instruction i_β in call graph node n_β if both access the same field of the same abstract object, at least one of the instructions is a write access, and $\langle n_\alpha, i_\alpha \rangle$ may happen in parallel with $\langle n_\beta, i_\beta \rangle$.

5.2 May happen in parallel

We now introduce an approximation of the happens-in-parallel relation induced by the `AsyncTask`. For simplicity, we present the algorithm from the perspective of analyzing the races involving one `AsyncTask` at a time.

Let n_b be the abstract call graph node for the analyzed `doInBackground` method. Let n_h be the event handler call graph node which executed n_b's `AsyncTask`– note that, depending on the choice of abstraction, there can be multiple call graph nodes representing runtime invocations of `doInBackground`. Let N_h be the set of all the event handler call graph nodes in the current application. For the example in Fig. 3, n_b is the invocation of the `doInBackground` method on line 15, and n_h is the execution of `onLoadFinished` (line 2) which led to n_b.

Let i_e be the instruction which executes the `AsyncTask` containing n_b. For our example in Fig. 3, i_e is `execute` method invocation at line 19. Let n_e be the node executing i_e. Our choice of context sensitivity ensures its uniqueness.

Let G^* be the so called supergraph [32] having as nodes pairs $\langle n, i \rangle$, where n is an call graph node, and i is an instruction in n. Intra-procedural, i.e., control flow graph (CFG), edges are lifted to the new graph. Call site instructions are linked to the entry and from the exit nodes of the lifted CFG corresponding to the call's target call graph node.

Let $G^*_{c \not\to r}$ be G^* with all its call-to-return edges removed. Call-to-return edges are the intra-procedural edges linking a call instruction directly to the next instruction, bypassing the interprocedural paths. Figure 5 shows the supergraph without call-to-return edges for the example in Fig. 3. Removing call-to-return edges does not affect reachability but it does affect the dominator relation used below. Call-to-return edges prevent instructions in a called method dominate any instruction after the call.

We say that the instruction $\langle n_\alpha, i_\alpha \rangle$ *may happen in parallel* with instruction $\langle n_\beta, i_\beta \rangle$ if $\langle n_\alpha, i_\alpha \rangle$ is reachable from the `doInBackground` node n_b, and either $\langle n_\beta, i_\beta \rangle$ does not dominate $\langle n_e, i_e \rangle$ on $G^*_{c \not\to r}$, or, if n_β calls n_e, i_β does not dominate the call to n_e on the n_β's CFG. E.g., in Fig. 5, $\langle n_\alpha, i_\alpha \rangle$ is the node for line 17 which is within the `doInBackground` method. $\langle n_\beta, i_\beta \rangle$ is the node for line 4, which does not dominate the forking node (line 19). Thus, the nodes for lines 4 and 17 may happen in parallel.

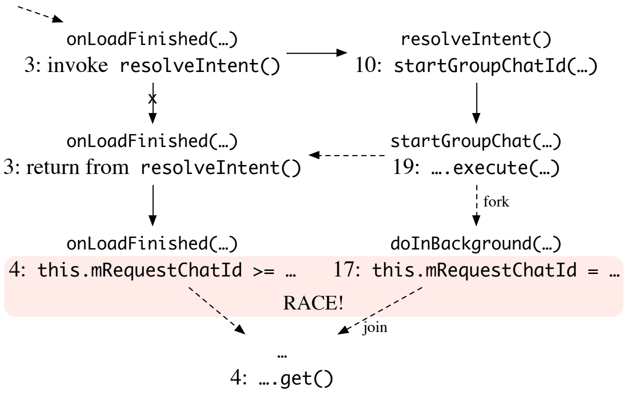

Figure 5: Supergraph without call-to-return edges ($G^*_{c \not\to r}$) for the code snippet in Fig. 3. The nodes are call graph node-instruction pairs. The arrows are intra and inter procedural edges. The crossed-out arrow is part of G^* but not $G^*_{c \not\to r}$. Dashed arrows denote *reachable* relations.

Furthermore, as the two instructions read and write the same field (`mRequestChatId`) of the same object (`this`), they may race. Thus, our tool raises a warning.

5.3 Android model

Android applications are event-based so exercising the code depends on events triggered by the UI, sensors, network, etc. In order to analyze the application statically, ASYNCHRONIZER uses a synthetic model of several key Android classes.

Figure 6 shows the callgraph for the code snippet in Fig. 1. ASYNCHRONIZER creates synthetic calls between the object initializer (the bytecode `<init>` method called before the constructor) of an activity or widget and its events handlers. Thus, `MainActivity`'s initializer calls, among others, its `onOptionsSelected` event handler. Similarly, ASYNCHRONIZER puts a synthetic call between a listener's initializer node to its handlers, and between the an `AsyncTask`'s `execute` and its `doInBackground`. This is an over-approximation of the application's possible behavior because it may be possible that a particular event will not be triggered. As the analysis is flow insensitive, it does not matter that the handler method is invoked at the handler object initialization point, not at the event trigger point.

ASYNCHRONIZER use the following strategy to select entry point for the analysis: *(1)* if the refactored class itself is an activity, it uses `Activity.<init>` as entry point; *(2)* if the refactored class is a GUI widget class (i.e., a `View`), it uses the object initializer of both the activities who use this widget, and the widget class itself as entry point (i.e., the analysis may run multiple times).

In terms of safety, our analysis is subject to the traditional limitations of static pointer analysis. Aside from the

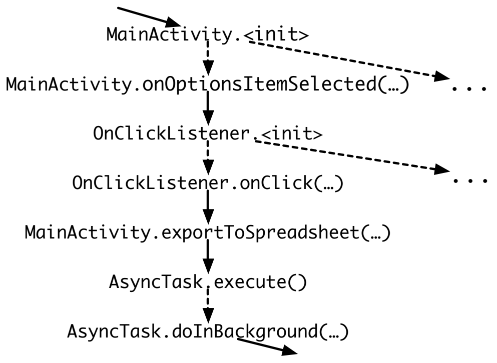
MainActivity.<init>
MainActivity.onOptionsItemSelected(…)
...
OnClickListener.<init>
OnClickListener.onClick(…)
...
MainActivity.exportToSpreadsheet(…)
AsyncTask.execute()
AsyncTask.doInBackground(…)

Figure 6: Part of the callgraph for the code in Fig. 1. Dashed arrows are synthetic call graph edges.

synthetic calls described above, reflection and native code are only handled up to what is provided by the underlying pointer analysis engine, WALA [10]. In some cases, Android apps use reflection to construct objects such as GUI widgets. Our analysis does not analyze such objects. This could be improved by looking into the configuration files used for defining the UI [38]. Also, the analysis' call graph contains a single node for each event. Considering our may happen in parallel definition, this may lead to false negatives for the cases where an event is invoked repeatedly.

Regarding precision, our current implementation does not consider event handling order. This leads to some false warnings in our evaluation. For example, the `onCreate` handler is always handled before `onStart`. Thus, an `AsyncTask` started in `onStart` could not happen in parallel with `onCreate`.

6. EVALUATION

To evaluate the usefulness of ASYNCHRONIZER we answer the following evaluation questions:
EQ1. Applicability: How applicable is ASYNCHRONIZER?
EQ2. Accuracy: How accurate is ASYNCHRONIZER when performing the code transformations?
EQ3. Effort: How much programmer effort is saved by ASYNCHRONIZER?
EQ4. Safety: Is ASYNCHRONIZER safer than manual refactorings?
EQ5. Value: Do programmers find refactorings applied by ASYNCHRONIZER useful?

6.1 Experimental Setup

We want to evaluate ASYNCHRONIZER on real open-source code, but because we are not the original developers of the code, it is hard to know on which code to apply the refactoring. Thus, we use two sets of experiments. First, we let the source code itself tell us which parts need to be refactored. To do this, we run ASYNCHRONIZER on projects that were manually refactored by the open-source developers, and compare the outcomes. Second, we start from the responsiveness issues detected by other researchers [37] and run ASYNCHRONIZER on code that was not refactored yet and determine whether the refactorings are useful for the original code developers. We use the first experiment to answer EQ1–EQ4, and the second experiment to answer EQ5.

Replicating existing refactorings. We select 13 projects from the 104 projects that we used in our formative study. Each of these 13 projects has at least two manual refactorings from sequential code to concurrent code via `AsyncTask`. The left-hand side of Tab. 2 shows the size of each project in non-blank, non-comment source lines of code[1]. For each project, we applied ASYNCHRONIZER to the code version just before the version that contained manual refactorings, and we only refactored the same code as the manual refactorings did. Notice that manual refactorings occur in several versions, so we checked out the version we need every time we applied ASYNCHRONIZER. We applied ASYNCHRONIZER to replicate all 77 manual refactorings in these 13 projects.

We report several metrics for each project. To measure the applicability, we count how many code fragments met the refactoring preconditions and thus can be refactored by ASYNCHRONIZER.

To measure the accuracy of code transformations, we compared the code transformed by ASYNCHRONIZER with manually changed code, and report the number of cases that have differences in `doInBackground` or `onPostExecute` method. Notice that here we are only interested to compare the code changes (described in Sec. 4), but these changes may still contain data races (we answer safety separately).

To measure the effort that a programmer would spend to refactor the code manually, we report the number of lines and files that are modified by the refactoring. These numbers are a gross estimate of the programmer effort that is saved when refactoring with ASYNCHRONIZER. Although we measure effort indirectly, many changes and analysis are non-trivial.

To answer the safety question, we ran ASYNCHRONIZER to analyze data races introduced by transformation. Notice that we bubbled up the races in libraries to application level [30]. We manually checked all the races and categorize them into four categories: *(fixed directly)* the races are fixed by developers during their manual refactoring; *(fixed later)* the races are not fixed during manual refactoring, but are fixed in a later version; *(never fixed)* the races are not fixed even in the latest version; *(false)* the races are false warnings.

The races that are fixed directly manifested immediately after a developer first encapsulated code into `AsyncTask`. Since in their commit the developers included both the refactoring and the race fixes, it implies that they are aware of the existence of these races. For the races that are fixed later, we also count how many days on average it took developers to find and fix races, as reported by the time span between the commit that introduces the race and the commit that fixes the races. For the races that are still not fixed in the latest version, we reported all of them to developers and suggested how to fix them.

Applying new refactorings. The preferred way to test for responsiveness is to run performance tests. However, none of the Android apps that we found had performance tests. Creating performance tests requires domain knowledge, generating test inputs that are representative (e.g., relevant database entries), etc. Thus, to measure the value of the refactoring, we select six projects that have potential responsiveness issues (shown in Tab. 3). These issues are detected in [37] but they have neither been reported, nor fixed.

Notice that these six projects are different from the 104 projects in our formative study. We manually identified the latent long-running operations in main thread. For example,

[1]We use David Wheeler's SLOCCount [8] to get size and we only report size of Java code.

Table 2: Results of applying Asynchronizer to 13 projects that have manual refactorings.

Project Name	SLOC	Applicability		Accuracy	Effort		Safety					
		Passed	Failed	Diff. #	LOC Mod.	Files Mod.	fixed directly	fixed later	never fixed	false	Total	Fix Time (day)
LibrelioDev-Android	15120	4	2	1	212	5	0	16	0	15	31	54
Sonet	18294	14	0	0	708	4	0	0	12	20	32	–
SocializeSDK-Android	65032	3	1	0	191	4	0	0	0	2	2	–
ChatSecureAndroid	35220	3	0	0	187	3	5	0	7	7	19	–
GwindowWhatAndroid	8785	4	0	0	68	4	0	0	0	0	0	–
Irccloud-Android	21384	3	0	0	82	3	3	8	3	0	14	124
Cyclestreets-Android	13523	3	0	0	98	3	0	0	0	0	0	–
Owncloud-Android	17016	3	0	0	108	3	4	0	0	8	12	–
AndroidSettings	71226	5	0	0	117	4	0	0	0	0	0	–
AndroidCalendar	34090	4	0	0	113	4	0	0	0	0	0	–
MyExpenses	20914	2	0	0	34	2	0	0	0	0	0	–
Allplayers-Android	5693	21	0	0	361	18	3	12	0	5	20	1
GRTransit	3316	4	1	0	206	5	0	2	2	0	4	14
Total	329613	73	4	1	2394	62	15	38	24	57	134	193

we search for call sites to database APIs in main thread. Then, we applied Asynchronizer on these operations and generated patches from the refactoring. When Asynchronizer raised a race warning, we checked and fixed the race. We also included the fix in the refactoring patches. We submitted these patches to developers. In total, we applied Asynchronizer to 58 places (column `Passed` + `Failed` in Tab. 3) in these projects. We grouped all changes and submitted six patches (one patch per project).

6.2 Results

Table 2 tabulates results of applying Asynchronizer to 13 projects that have manual refactorings.

Applicability: Columns 3 and 4 show the number of refactorings that pass or fail preconditions P1–P3. Among the 77 places where we applied the refactoring, 73 satisfy the preconditions. This shows that our refactoring is highly applicable.

Of the four places that fail preconditions, 3 failed P1, 1 failed P2, 1 failed P3 (one case failed two preconditions). We had to manually modify the code to satisfy the preconditions. To satisfy precondition P1, we convert the local variables into fields. For precondition P2, we temporarily remove the return statements before refactoring and put them back to the appropriate places after refactoring. For precondition P3, we expanded the selected expression into a full statement, and then supplied it as the input to Asynchronizer.

After changing the input source code to pass preconditions, we applied Asynchronizer to these four cases and included them along with the other metrics shown in Tab. 2.

Accuracy: Column 5 shows the number of differences between manual and automated refactorings. The differences do not include other changes made by developers (e.g., adding new features). There is only one case in `LibrelioDev-Android` project where the two outputs differ. In this case, manual refactoring moves fewer statements into `onPostExecute` handler, but they don't affect the semantics, which means that the code behaves the same way in both cases.

Effort: In total, the refactoring modified 2394 lines of code in 62 files (see `LOC Mod.` and `Files Mod.` columns in Tab. 2). On average, each refactoring changes 31 lines of code. More important, many of these changes are non-trivial: programmers need to infer fields, method parameters, and return value, which statements can be moved into `onPostExecute`, as well as deal with special cases. In contrast, when using Asynchronizer, the programmer only has to initiate the transformation. Asynchronizer takes less than 10 seconds per refactoring.

```
public class OAuthLogin extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        ...
        new AsyncTask<String, Void, String>() {
            protected String doInBackground(String... args) {
                mOAuthProvider = new CommonsHttpOAuthProvider(...);
            }
        }.execute(...);
    }
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        ...
        new AsyncTask<String, Void, String>() {
            protected String doInBackground(String... args) {
                mOAuthProvider.retrieveAccessToken(...);
            }
        }.execute(...);
    }
}
```

Figure 7: Developers of `Sonet` project confirmed that the races on `mOAuthProvider` will not lead to bug.

Safety: Columns 8 to 12 show the 134 races that Asynchronizer detected automatically and we checked manually.

Notice that 38 races are not fixed immediately in the manual refactoring, but are fixed in a later version. The strategies to fix these races include adding synchronizations, moving the statements involved in races outside of `AsyncTask`, changing shared variables into local variables, or removing the shared variables. Interestingly, among these 38 races, 12 races in `Allplayers-Android` project are fixed incorrectly the first time: developers invoke `get` immediately after executing the `AsyncTask`. They applied a second patch to fix them correctly in a later version. In the four projects that fix races in a later version, developers spent 193 days in total to apply patches (`Fix Time` column). There are 57 false warnings.

Table 3: Results of applying Asynchronizer to 6 projects that have potential responsiveness issues.

Project Name	SLOC	Passed	Failed	Races	Files Mod.
Connectbot	33326	24	0	70	5
FBReaderJ	58718	10	0	8	5
K-9 Mail	78679	7	0	38	5
KeePassDroid	28588	1	2	0	2
Vudroid	2408	1	0	13	1
VLC	36852	13	0	16	8
Total	238571	56	2	145	26

The reasons for the false warnings were discussed in Sec. 5.3.

The remaining 24 races still exist in the latest version. We reported all of them to developers. They fixed 3 races in `Irccloud-Android`, and they confirmed 9 races in `ChatSecureAndroid` and `GRTransit`. The developers of `Sonet` do not think the 12 races lead to bugs. We show one simplified code snippet from `Sonet` in Fig. 7. In the experiment, we think the two `AsyncTask` (lines 4 and 12) can be executed concurrently, so there is a race on `mOAuthProvider` (lines 6 and 14). However, the developers confirmed that the second task cannot run until the first completes. In practice, developers can avoid checking such races by customizing synthetic call graphs based on their domain knowledge about which event handlers may happen in parallel. For this example, they can remove the second event handler from the call graph, while checking the first task.

Our result shows 62 (columns `fixed later` + `never fixed`) out of 134 races are neither detected nor fixed when developers perform manual refactoring. Even when they are fixed in a later version, the timespan is long. Thus, ASYNCHRONIZER is safer than manual refactoring.

Value: Table 3 shows results where we used ASYNCHRONIZER to refactor long-running operations from main thread into `AsyncTask`. We used a corpus of 6 projects, where in total we applied ASYNCHRONIZER to 58 places in 26 files. 56 cases satisfied the preconditions. Similar to the previous experiment, for the two cases that failed the preconditions, we manually modified the code to satisfy the preconditions. We also check the races reported by ASYNCHRONIZER (column 5). Notice that the races we show in Tab. 3 do not include false warnings (there are 72 false warnings in total).

We created patches which include the transformations and fixes for races, and submitted the patches to the open-source developers. At the time when this paper is written, the developers from `K-9 Mail` and `KeePassDroid` have accepted ten refactorings. The developers of `Vudroid` and `VLC` do not think the operations encapsulated into `AsyncTask` significantly affect UI responsiveness. For example, `Vudroid` developers said "*`ZoomRoll` class instance is a singleton for application and hence your patch will change only the first time load delay. I have never observed considerable time delays on Activity start*". This shows the importance of having domain knowledge, but also shows that our refactoring approach can produce useful results accepted by developers.

7. RELATED WORK

Liu et al. [27] empirically study three performance bug patterns in Android apps, one of which is running long-running operations in main thread. They also propose an approach to detect such operations statically. Yang et al. [37] test the responsiveness of Android apps by adding a long delay after each heavy API call. Choi et al. [17] uses machine learning to learn a model for smartphone apps and generate test inputs from the model. Jensen et al. [21] propose a test generation approach to find event sequences that reach a given target line in smartphone app. Concolic testing [12] and random testing [20] is also applied to smartphone apps. Their work focuses on testing of Android apps. However, our work is a compensation of theirs: developers can exploit `AsyncTask` refactoring to eliminate the performance issues that are detected in testing.

Sai et al. [38] propose an approach to detect GUI accesses from outside the main thread. This technique can be integrated with ASYNCHRONIZER to help developers avoid GUI access in `AsyncTask` while performing refactoring. Some techniques are proposed to analyze event-driven applications. Raychev et al. [31] proposed a dynamic data race detection for event-driven program based on Javascript. Zheng et al. [39] statically check concurrency bugs due to asynchronous calls for Ajax applications. Less scalable, model-checking based, techniques have also been proposed for event-driven or GUI applications [14, 19, 35].

There are several empirical studies on the usage of libraries, software evolution and refactoring [15, 16, 22–24, 26, 28, 29]. Kavaler et al. [22] study how Android APIs are used and asked by developers. Kim et al. investigate API-level refactorings [23] and benefits from refactoring [24]. Bavota et al. [15] study the bugs introduced by refactorings. Our previous work [26, 29] shows people tend to misuse concurrent APIs in Java and C#. In this work, we study how people use concurrent APIs provided by Android framework. Several refactorings are proposed to improve performance by introducing concurrency [18, 33] or optimizing binary code [34], while others guarantee the correctness of concurrent program [25, 36]. However, our refactoring retrofit concurrency into Android apps to improve responsiveness.

8. CONCLUSIONS

Developers introduce concurrency into programs via concurrent constructs. However, refactoring sequential code to concurrent code is non-trivial, tedious, and error-prone.

We presented a formative study on Android's `AsyncTask`. Our study shows that developers use `AsyncTask`, both to implement new features, and to refactor existing sequential code. However, we found that manual refactoring introduces performance bugs: by misplacing the `AsyncTask.get` their "concurrent" code runs sequentially. Also we found data races in manually refactored code. In some cases, it took developers hundreds of days to find and fix these bugs.

We presented ASYNCHRONIZER, which automates refactoring sequential code to use `AsyncTask`. The refactoring is composed of two steps: a code transformation that moves user-selected code into `AsyncTask`, and a safety analysis that checks data races. In our empirical evaluation we applied ASYNCHRONIZER on 19 Android apps. We found that the tool is widely applicable, it is accurate compared to manual code transformations, it saves programmers' effort, it is safer than manual refactoring, and open-source developers accepted several patches with refactorings created by ASYNCHRONIZER. This shows that the ASYNCHRONIZER is useful.

9. REFERENCES

[1] Android Processes and Threads. http://developer.android.com/guide/components/processes-and-threads.html.

[2] Asynchronizer. http://mir.cs.illinois.edu/~yulin2/asynchronizer.

[3] Eclips Refactoring Engine. https://www.eclipse.org/articles/Article-LTK/ltk.html.

[4] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/.

[5] GiTactive. https://github.com/kevinsawicki/gitective.

[6] GitHub. https://github.com.

[7] JDK Swing Framework. http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[8] SLOCCount. http://www.dwheeler.com/sloccount/.

[9] The SWT Ttoolkit. http://eclipse.org/swt/.

[10] T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php.

[11] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[12] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 1–11, 2012.

[13] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[14] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. In *Proc. of the Conference on Software for Citical Systems*, SIGSOFT '91, pages 16–28, 1991.

[15] 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 *Proc. of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 104–113, 2012.

[16] N. Cacho, E. Barbosa, T. Cesar, A. Garcia, T. Filipe, and E. Soares. Trading robustness for maintainability: An empirical study of evolving C# programs. In *Proc. of the 36th International Conference on Software Engineering*, ICSE '14, To Appear.

[17] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proc. of the 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, 2013.

[18] D. Dig, J. Marrero, and M. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proc. of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, 2009.

[19] M. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *Proc. of the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '97, pages 244–261, 1997.

[20] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proc. of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, 2011.

[21] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proc. of the 13th International Symposium on Software Testing and Analysis*, ISSTA '13, pages 67–77, 2013.

[22] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. T. Devanbu, and V. Filkov. Using and asking: APIs used in the android market and asked about in stackoverflow. In *SocInfo*, volume 8238 of *Lecture Notes in Computer Science*, pages 405–418, 2013.

[23] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of API-level refactorings during software evolution. In *Proc. of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, 2011.

[24] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, 2012.

[25] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Transformation for class immutability. In *Proc. of the 33rd International Conference on Software Engineering*, ICSE '11, pages 61–70, 2011.

[26] Y. Lin and D. Dig. Check-then-act misuse of java concurrent collections. In *Proc. of the 6th International Conference on Software Testing, Verification and Validation*, ICST '13, pages 164–173, 2013.

[27] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proc. of the 36th International Conference on Software Engineering*, ICSE '14, To Appear.

[28] S. Okur and D. Dig. How do developers use parallel libraries? In *Proc. of the 20th ACM SIGSOFT International Symposium on Foundations of software engineering*, FSE '12, 2012.

[29] S. Okur, D. Hartveld, D. Dig, and A. Deursen. A study and toolkit for asynchronous programming in C#. In *Proc. of the 36th International Conference on Software Engineering*, ICSE '14, To Appear.

[30] C. Radoi and D. Dig. Practical static race detection for java parallel loops. In *Proc. of the 13th International Symposium on Software Testing and Analysis*, ISSTA '13, pages 178–190, 2013.

[31] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proc. of the 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 151–166, 2013.

[32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[33] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Refactoring java programs for flexible locking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 71–80, 2011.

[34] E. Tilevich and Y. Smaragdakis. Binary refactoring: Improving code behind the scenes. In *Proc. of the 27th International Conference on Software Engineering*, ICSE '05, pages 264–273, 2005.

[35] O. Tkachuk and M. Dwyer. Environment generation for validating event-driven software using model checking. *IET Software*, 4(3):194–209, 2010.

[36] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *Proc. of the 7th ACM SIGSOFT Symposium on The Foundations of Software Engineering*, FSE '09, pages 173–182, 2009.

[37] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in android applications. In *Proc. of International Workshop on the Engineering of Mobile-Enabled Systems*, MOBS '13, pages 1–6, 2013.

[38] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded gui applications. In *Proc. of the 12th International Symposium on Software Testing and Analysis*, ISSTA '12, pages 243–253, 2012.

[39] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proc. of the 20th International Conference on World Wide Web*, WWW '11, pages 805–814, 2011.