Refactorings for Android Asynchronous Programming

Yu Lin
Computer Science Department
University of Illinois at Urbana-Champaign, USA
yulin2@illinois.edu

Danny Dig
School of Electrical Engineering and Computer Science
Oregon State University, USA
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 provides several async constructs that developers can use, developers can still miss opportunities to encapsulate long-running operations in async constructs. On the other hand, they can use the inappropriate async constructs, which result in memory leaks, lost results, and wasted energy. Fortunately, refactoring tools can eliminate these problems by retrofitting asynchrony to sequential code and transforming async code to use the appropriate constructs. This demo presents two refactoring tools for Android apps: (i) ASYNCHRONIZER, a refactoring tool that enables developers to extract long-running operations into Android `AsyncTask`. (ii) ASYNCDROID, a refactoring tool which enables developers to transform existing improperly-used `AsyncTask` into Android `IntentService`.**

I. 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 jarring than when using a mouse.

Previous research [29], [37] shows that many Android apps suffer from poor responsiveness. One of the primary reasons for unresponsiveness is executing all workload in the UI event thread. The UI event thread of an Android app processes UI events, but long-running operations (i.e., blocking I/O operations) will "freeze" it, so that it cannot respond to new user input or redraw.

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 demo, we first present an automated refactoring tool, ASYNCHRONIZER, 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. 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 [31] race detector.

Our study [27] shows `AsyncTask` is the most widely used async construct in Android. However, `AsyncTask` is designed for encapsulating short-running tasks (i.e., less than one second) and if improperly used, can lead to memory leaks, lost results, and wasted energy. 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 developer forums [5], [7], [8], this problem is widespread and is critical for long-running tasks. Android provides another async construct, `IntentService`, for long-running tasks. `IntentService` does not have the above limitations because it does not hold a reference to GUI and instead uses a radically different mechanism to communicate with the GUI. To avoid the above problems of `AsyncTask`, developers must refactor long-running `AsyncTask` code into enhanced async constructs such as `IntentService`.

In this demo, we also present ASYNCDROID, an automated refactoring tool that 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.

ASYNCHRONIZER is available at: http://refactoring.info/tools/asynchronizer and ASYNCDROID is available at: http://refactoring.info/tools/asyncdroid

```
1   public class RouteselectActivity extends Activity {
2     ...
3     private void startManagingCursor(Cursor csr, int count) {...}
4     public void onCreate() {
5       ...
6       ListView lv = getListView();
7       final String qry = "select...";
8
9
10
11
12
13
14
15
16      Cursor mCsr = DatabaseHelper.ReadableDB().rawQuery(qry);
17      int count = mCsr.getCount();
18      startManagingCursor(mCsr, count);
19
20
21
22      lv.setOnTouchListener(mGestureListener);
23      if (count > 1)
24        tv.setText(R.string.route_fling);
25      else if (count == 0)
26        tv.setText(R.string.stop_unused);
27  }}
```

(a) Synchronous Code

```
1   public class RouteselectActivity extends Activity {
2     ...
3     private void startManagingCursor(Cursor csr, int count) {...}
4     public void onCreate() {
5       ...
6       ListView lv = getListView();
7       final String qry = "select...";
8       RouteselectTask task = new RouteselectTask(lv);
9       task.execute(qry);
10    }
11    class RouteselectTask extends AsyncTask<String, Void, Integer>{
12      ListView lv;
13      RouteselectTask (ListView lv) { this.lv = lv; }
14      protected Integer doInBackground(String... args) {
15        String qry = (String) args[0];
16        Cursor mCsr = DatabaseHelper.ReadableDB().rawQuery(qry);
17        int count = mCsr.getCount();
18        startManagingCursor(mCsr, count);
19        return count;
20      }
21      protected void onPostExecute(Integer count) {
22        lv.setOnTouchListener(mGestureListener);
23        if (count > 1)
24          tv.setText(R.string.route_fling);
25        else if (count == 0)
26          tv.setText(R.string.stop_unused);
27  }}}
```

(b) Async Code with `AsyncTask`

Fig. 1: Relevant code from GR-Transit app. Programmer selects lines 16 to 18 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.

II. THE ASYNCHRONIZER TOOL

A. *Introduction to Android Programming and* `AsyncTask`

Android GUIs are typically composed of several *activities*. An activity represents a GUI screen. For example, the login screen of an email client is an activity. Similar to many other GUI frameworks such as Swing [12] and SWT [13], Android uses a **single thread model** to process events [6]. Events in Android apps include lifecycle events (e.g., activity creation), user actions (e.g., button click), sensor inputs (e.g., orientation change), etc. Developers define event handlers to respond to these events. When an application is launched, the system creates a *main thread*, in which it will run the event handlers. Figure 1(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. `onCreate` is an event handler that is invoked when an activity is created. It queries database (lines 16 to 18) and show the query results on GUI (lines 22 to 26).

However, blocking calls such as querying database can block the main thread, so developers should use async programming to avoid "freezing" the GUI. To ease the use of asynchrony, Android framework provides `AsyncTask`. `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.

Figure 1(b) shows the corresponding async code of Fig. 1(a), with the use of `AsyncTask`. The methods that start with "`on`" are event handlers. `onCreate` handler creates an `AsyncTask` (lines 8 and 9) and executes it concurrently with the main thread. The `doInBackground` method (line 14) encapsulates the work that executes in the background. The task contains the database query. Finally, the result of the background computation is returned for main thread to use (i.e., `count` at line 19). The main thread executes the `onPostExecute` handler after `doInBackground` finishes to update GUI (lines 22 to 26).

B. *Refactoring Workflow and Preconditions*

We implemented ASYNCHRONIZER as a plugin in the Eclipse IDE [9]. 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. In Fig. 1(a), if the programmer applies our transformation to lines 16 to 18, ASYNCHRONIZER will transform the code to Fig. 1(b).

After the transformation, the programmer can invoke ASYNCHRONIZER's safety analysis component to check data races due to the transformation. If ASYNCHRONIZER found data races, the programmer still needs to confirm and fix them manually. Only after this the refactored code is correct.

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 [1].

[1]An improvement is to declare multiple return variables as fields of the `AsyncTask` instead of local variables. This way can eliminate **P1**.

(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. Similarly, the `break` and `continue` statements are only allowed if they are selected along with their enclosing loop.

(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 block and wait for the task result; this defies the whole purpose of launching an `AsyncTask`.

C. Analysis and Transformation for `AsyncTask`

Create the `doInBackground` Method. The first step of the transformation is to move the selected statements into the `doInBackground` method. This is similar to EXTRACT METHOD refactoring. In this step, ASYNCHRONIZER needs 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. 1(b), the `doInBackground` method takes one arguments, `qry` and returns `count`.

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. Otherwise, the refactored program will have a different semantics. The algorithm for creating `onPostExecute` can be found in [28]. In the example shown in Fig. 1(a), all the statements after the selected code (lines 22 to 26) can be put into `onPostExecute`.

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. 1(b), local variable `lv` is needed by `onPostExecute`. ASYNCHRONIZER declares a field `lv` in the `AsyncTask` (line 12) and adds a constructor to initialize this field (line 13). After that, it creates an inner class declaration using all the code elements which have been created above (line 11). Finally, it generates two statements to create task instance and call `execute` method, and replaces the selected code by these two statements (lines 8 and 9).

Safety Checking for the Refactoring. Our study [28] 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 [31]. 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. 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 extended ITERACE to find races that occur between `doInBackground` and other threads and integrated it with ASYNCHRONIZER. Details can be found in [28].

Summary of Evaluation. To evaluate ASYNCHRONIZER's usefulness, we used it to refactor 200 places in 32 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. Thus, it can save programmers' effort. 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 123 refactorings in 19 apps. 10 replied and accepted 40 refactorings. This shows that ASYNCHRONIZER is valuable.

III. THE ASYNCDROID TOOL

A. Introduction to `IntentServices` *in Android*

As shown in Sec. I, `AsyncTask` is only fit for short-running tasks (i.e., less than one second), since it can lead to memory leaks, lost results, and wasted energy. For long-running tasks, developers should use `IntentService`. `IntentService` encapsulates operations that are executed in a background thread instead of main thread. Unlike `AsyncTask` which use shared-memory for communication, `IntentService` uses a distributed-style programming to communicate with main thread.

Figure 2 shows an equivalent implementation of Fig. 1(b) using `IntentService`. Service is started when `startService` is invoked (line 12). The background task from Fig. 1(b) is now put into `onHandleIntent` method (line 29). `onHandleIntent` method is executed in a background thread. After the task is finished, `IntentService` sends its task result via `sendBroadcast` method (lines 36). To get the task result and update GUI, the GUI should declare and register a broadcast receiver using a broadcast filter (lines 2 and 9). Once the registered receiver receives (i.e., the GUI) this broadcast, its `onReceive` method will be executed on main thread. For example, a receiver is declared at line 14 and registered at line 9, and its `onReceive` get the task result and updates GUI (lines 19 to 24). Thus, `onReceive` is semantic-equivalent to `onPostExecute` in `AsyncTask`.

Notice that the objects transferred between `IntentService` and main thread (e.g., task result) should be wrapped by an `Intent` object, which is marshalled and sent via broadcast (lines 10, 11, 34, 35). `Intent` is analogous to a hash map whose key is a string and value is a serializable

```
1   public class RouteselectActivity extends Activity {
2     public static final FILTER = "Routeselect_receiver";
3     private RouteselectReceiver receiver;
4     protected void onCreate() {
5       ...
6       ListView lv = getListView();
7       final String qry = "select...";
8       receiver = new RouteselectReceiver(lv);
9       this.registerReceiver(receiver, new IntentFilter(FILTER));
10      Intent intent = new Intent(this, RouteselectService.class);
11      intent.putExtra("qry", qry);
12      this.startService(intent);
13   }
14   private class RouteselectReceiver extends BroadcastReceiver {
15     ListView lv;
16     int count;
17     public RouteselectReceiver(ListView lv){ this.lv = lv; }
18     public void onReceive(Context context, Intent intent) {
19       int count = intent.getStringExtra("RV");
20       lv.setOnTouchListener(mGestureListener);
21       if (count > 1)
22         tv.setText(R.string.route_fling);
23       else if (count == 0)
24         tv.setText(R.string.stop_unused);
25 }}}
26 public class RouteselectService extends IntentService {
27   String qry;
28   private void startManagingCursor(Cursor csr, int count) {...}
29   public void onHandleIntent(Intent intent) {
30     this.qry = intent.getStringExtra("qry");
31     Cursor mCsr = DatabaseHelper.ReadableDB().rawQuery(qry);
32     int count = mCsr.getCount();
33     startManagingCursor(mCsr, count);
34     Intent result = new Intent(RouteselectActivity.FILTER);
35     result.putExtra("RV", count);
36     sendBroadcast(result);
37 }}
```

Fig. 2: Async Code using `IntentService` that is semantic-equivalent with Fig. 1(b)

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. 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 9 and 34 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.

B. Refactoring Workflow and Preconditions

Our study [27] shows `AsyncTask` is being overused at the expense of `IntentService`. Inspired by these findings, we implemented 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 the refactoring as a plugin in the Eclipse IDE, on top of Eclipse JDT [10] and refactoring engine [9]. 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. It helps developers migrate the inappropriately used `AsyncTasks` to `IntentServices`. Additionally, 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`.

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.

Figure 1(b) is a valid example of a target program that meets all these preconditions.

C. The Refactoring Approach

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. An example of transferred object is the argument `qry` and the return value `count` in Fig. 1(b) (lines 9, 19).

As mentioned in Sec. III-A, the transferred objects are marshalled and required to be serializable. ASYNCDROID traverses the type hierarchy and checks the serializability of an variable type based on its definition [11]. 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 transferred objects 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 flows into `IntentService` (line 27 in Fig. 2). ASYNCDROID adds statements to unwrap objects from `Intent` at the beginning of `onHandleIntent` (line 30 in Fig. 2) . At every exit of `doInBackground`, ASYNCDROID creates an `Intent` to carry task results, and sends it via broadcast (line 36 in Fig. 2). Second, since `doInBackground` can invoke methods defined in `AsyncTask` or refactored class, ASYNCDROID moves (or copies if multiple call sites exist) such methods into `IntentService` class (line 28 in Fig. 2). Finally, ASYNCDROID rewrites the call sites of `AsyncTask.execute` into `startService`. This includes creating the `Intent` object to wrap the arguments (lines 10, 11 in Fig. 2).

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` (lines 18 in Fig. 2). ASYNCDROID also inserts statements at the beginning of `onReceive` to unwrap task results (lines 30 in Fig. 2). A filter is required when registering receiver. The filter specifies which broadcast a receiver can receive. ASYNCDROID concatenates refactored class name and receiver name as filter name, and uses it to register receiver and send broadcast (lines 2, 9 and 34 in Fig. 2). Notice that the receiver should be registered in lifecycle event handlers (e.g., `onCreate`) to avoid lost task results during GUI recreation. Details can be found in [27].

Summary of 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.

IV. DISCUSSION

Asynchrony is crucially important in mobile devices. With their rich array of sensors, camera, and GPS all integrated in a convenient form factor, mobile devices offer exciting new applications and services never before possible on consumer devices. These can be harnessed only if apps leverage asynchrony to ensure responsive behavior. Refactoring tools help mobile apps to keep responsive.

Manual refactoring is tedious and error-prone. We hope the refactorings tools we presented in this demo can help Android developers retrofit asynchrony more efficiently and save their programming efforts. On the other hand, the refactored code provide an positive example to beginners about how to use async constructs (such as `IntentService`). For tool builders, we hope our proposal gives them a hint on building refactorings for async constructs on other platforms (e.g., WindowsPhone). Researchers can investigate the approaches of integrating performance bug detection techniques with async refactorings, and propose novel frameworks for performance bug detecting and fixing. As one of our future work, we plan to integrate our refactorings with Google ShipShape [3], which is a static program analysis platform allows custom analyzers to plug in through a common interface. We believe such integration can increase the impact of these refactorings.

V. RELATED WORK

Google provides several tools for performance testing of Android apps [1], [2], [4]. MonkeyRunner [2] generates pseudo-random streams of user and system events. TraceView [4] is a performance profiler which analyzes the execution logs and shows profiling results through a graphical viewer. StrictMode [1] provides ways to detect lengthy operations in main thread through testing.

Several researchers proposed testing and analysis approaches for mobile apps [14], [15], [22], [24], [29], [36]. Berardinelli et al. [15] introduced a framework for modeling and analyzing the performance of context-aware mobile software systems. Liu et al. [29] proposed a tool to detect blocking API invocations in main thread statically. Yan et al. [36] proposed a test generation approach to detect memory leaks for Android apps. Such testing tools can be integrated with our refactoring tools to identify where to apply the refactorings. Recent work on dynamic race detectors [21], [30] proposed a causality model for Android apps which they used to infer happens-before relationships between events. These dynamic race detectors can combine with ASYNCHRONIZER to reduce false positives in race detection phase.

Several refactorings that aim to improve non-functional qualities (e.g., performance through parallelism and concurrency) have been proposed . Schafer et al. [33] proposed a refactoring for replacing Java built-in locks with more flexible locks. Wloka et al. [35] presented a refactoring for replacing global state with thread local state. Schafer et al. [32] examined whether classic refactorings can be safely applied to concurrent programs. Our group implemented several concurrency-related refactorings to improve *throughput*, *scalability* and *asynchrony* [16]–[20], [23], [25], [26], [28], [31], [34].

VI. CONCLUSIONS

Refactoring tools can help programmers retrofit asynchrony into mobile apps. This demo presents two refactoring tools for Android apps. ASYNCHRONIZER is a tool for retrofitting asynchrony into sequential code via a basic Android async construct, `AsyncTask`. ASYNCDROID converts share-memory style `AsyncTask` into an enhanced async construct, the distributed-style `IntentService`. Our preliminary experience with refactoring several Android projects shows that ASYNCHRONIZER and ASYNCDROID are applicable, accurate, efficient and produce valuable refactorings.

VII. ACKNOWLEDGMENTS

This research is partly funded through NSF CCF-1439957 and CCF-1442157 grants, a SEIF award from Microsoft, and a gift grant from Intel.

References

[1] "Google Play," August 2015, http://developer.android.com/reference/android/os/StrictMode.html.

[2] "MonkeyRunner," August 2015, http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[3] "ShipShape Repository," August 2015, https://github.com/google/shipshape.

[4] "Traceview," August 2015, http://developer.android.com/tools/debugging/debugging-tracing.html.

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

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

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

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

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

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

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

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

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

[14] 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.

[15] 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.

[16] D. Dig, "A refactoring approach to parallelism," *IEEE Software*, vol. 28, no. 1, pp. 17–22, 2011.

[17] 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.

[18] ——, "How do programs become more concurrent? A story of program transformations." in *IWMSE'11: International Workshop on Multicore Software Engineering*, 2011, pp. 1–8.

[19] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "ReLooper: refactoring for loop parallelism in Java," in *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, ser. OOPSLA Companion '09, 2009, pp. 793–794.

[20] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. FSE '13, 2013, pp. 543–553.

[21] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 326–336.

[22] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '13, 2013, pp. 67–77.

[23] 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.

[24] 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.

[25] 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.

[26] ——, "A study and toolkit of Check-Then-Act idioms of java concurrent collections," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 397–425, 2015.

[27] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, 2015.

[28] 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 '14, 2014, pp. 341–352.

[29] 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.

[30] P. Maiya, A. Kanade, and R. Majumdar, "Race Detection for Android Applications," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 316–325.

[31] C. Radoi and D. Dig, "Practical static race detection for Java parallel loops," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '13, 2013, pp. 178–190.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.