Refactoring for Asynchronous Execution on Mobile

Danny Dig, *Member, IEEE,*

Abstract—**To improve responsiveness, oftentimes developers use asynchronous programming. In the post-PC era, asynchronous programming is even more in demand because mobile and wearable devices have limited resources and access the network excessively. One contemporary development task is refactoring long-running, blocking synchronous code (e.g., accessing the web, cloud, database, or file system) into non-blocking asynchronous code. This paper educates the mobile app developer on the kinds of refactorings they need to perform to improve responsiveness, along with the obstacles of using asynchrony. We also present our formative studies on understanding the challenges that developers face when retrofitting asynchrony, our program analyses and transformations together with our growing, practical toolset and resources that enable app developers to retrofit asynchrony.**

Index Terms—**refactoring, asynchronous programming, program analysis.**

I. Introduction

ASYNCHRONOUS programming is in demand today. Asynchrony is essential for I/O activities that are potentially blocking (e.g., accessing the web or the file system), or for long-running CPU activities (e.g., image encoding/decoding). Asynchrony helps an application to stay responsive because the application can continue with other work.

Asynchrony is especially valuable for applications that access the UI thread. Today's UI frameworks are usually designed around the use of a single UI event thread: every operation that modifies UI state is executed atomically as an event on that thread. The UI "freezes" when it cannot respond to input, or when it cannot be redrawn, and because it seems non-responsive, the user might become frustrated. It is recommended that long-running CPU-bound or blocking I/O operations execute asynchronously so that the application continues to respond to UI events.

Other times, asynchrony is the natural programming model. For example, event-driven programming models arise naturally to match the asynchronicity of input streams coming from diverse sources such as touchscreen, accelerometer, microphones, and other sensors on smartphones. Similarly, modern web applications make extensive use of asynchrony, via AJAX requests, and also of asynchronous code loading to reduce perceived page load time.

Without any loss of generality, in the rest of this paper we will use the domain of mobile devices as examples of asynchronous programming (other domains like desktops/laptops and web have the same asynchronous constructs). We focus on mobile apps because we expect to find many exemplars of asynchronous programming, given that responsiveness is critical. Mobile apps can easily be unresponsive [1] 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. Moreover, mobile apps are becoming increasingly more important. End users are rapidly moving from the desktop to the mobile world. According to Gartner [2], by 2016 more than 300 billion apps will be downloaded annually.

D. Dig is with the School of of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, 97331 USA e-mail: digd@eecs.oregonstate.edu.

There are several key problems that programmers encounter when working with asynchrony:

Lack of methods and tools for converting synchronous to asynchronous code. In our formative studies based on hundreds of open-source apps, we found that half of all asynchrony usages in real-world apps have not been introduced from scratch, but have been converted from previously existing synchronous code. This step is a source-to-source transformation that needs to preserve the current functionality of the application (albeit improve some non-functional property such as responsiveness), thus it is a *refactoring* [3]–[6]. This refactoring requires complex code transformations that invert the flow of control and introduce *callbacks* to notify the caller when an asynchronous operation is finished. Also, the refactoring requires reasoning about non-interference of asynchronous operations with the main thread of execution, otherwise the asynchrony can lead to non-deterministic data races. In our preliminary studies [7], [8] we have found, reported, and submitted hundreds of accepted patches against data-races caused by manual refactoring for asynchrony in Android and C# applications.

Currently, developers solve such problems and carry such non-trivial transformations manually. However, interactive refactoring tools can help.

Lack of clear knowledge of how to convert synchronous into asynchronous code. There are extensive programmer documentation (e.g., Android Best Practices for Performance [9]) or tools that detect I/O blocking operations (e.g., StrictMode [10] for Android). But they primarily focus on designing async programs from scratch. Thus, many existing programs suffer from poor responsiveness. A recent award-winning ICSE'14 paper [1] found that 75% of performance bugs in Android arise because of missed opportunities to introduce asynchrony in UI code. In our own preliminary results [7], [8] we discovered hundreds of places in UI code where asynchrony should have been used. We suggested 10 such places to the developers of four projects, and they subsequently changed their code to use asynchrony. This shows the need for educational resources on how to *retrofit asynchrony*.

We started to address some of these challenges by releasing educational resources, such as http://learnasync.net, that show thousands of examples of correct and incorrect usage of asynchrony in real-world apps. On the tooling side, we have been developing automated refactorings for converting synchronous into asynchronous code. We implemented refactorings that improve responsiveness of mobile apps by extracting long-running CPU or I/O blocking code into asynchronous code. We explored different flavors of asynchrony, such as framework-based callbacks in the Android SDK, task-based futures and continuations, etc.

Moreover, we also implemented refactorings for modernizing legacy asynchronous code. For example, one of our refactorings replaces the legacy style of asynchrony in the 10-year old Asynchronous Programming Model (APM) from .NET 1.0 with modern asynchronous language constructs such as the `async/await` from the most recent version of .NET.

The interested reader can learn more and download our growing toolset for refactoring at: http://refactoring.info/tools

II. Examples of Refactorings for Asynchrony

We will first introduce the asynchrony terminology, then we will use examples from the domain of GUI programming on mobile devices to illustrate different refactoring flavors and the refactorings that we support. Although our code examples use C# and .NET APIs, similar constructs already exist or are planned for Java and Android.

Most GUI frameworks such as the ones in .NET, Android, iOS, etc., use an event-driven model. Events in mobile apps include lifecycle events (e.g., GUI screen creation), user actions (e.g., button click, menu selection), sensor inputs (e.g., GPS, screen orientation change), etc. Developers define event handlers to respond to these events.

GUI frameworks use a **single thread model** to process events [9]. 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 screen pages. The main thread puts events into a single event queue, dequeues events, and executes corresponding event handlers.

When a GUI event handler executes a synchronous long-running CPU-bound or blocking I/O operation, the user interface will freeze because the UI event thread cannot respond to events. Consider the example in Fig. 1a which shows a handler that responds to a button click by downloading the content of the entry webpage of IEEE Computer Society and displaying it. Notice that the main thread invokes a blocking, potentially long-running operation, `getResponse`, which downloads the content of the webpage, and the UI will become unresponsive while the download is in progress. Fig 2a shows the execution flow for this code, and highlights the time when the UI is frozen during the page download.

To avoid unresponsiveness, programmers exploit asynchrony by encapsulating and running blocking I/O or long-running CPU computations in the background.

Next, we present the two prevalent styles of asynchronous programming. In Sec. II-A we present the framework-based, callback-based style, and in Sec. II-B we present the new pause-and-play style based on `async/await` language constructs. Our refactoring toolset targets both styles.

A. Refactoring from synchronous to callback-based asynchrony

Figure 1b shows the refactored, asynchronous version of the code in Fig. 1a. The .NET framework introduced a callback-based Asynchronous Programming Model (APM), in its first version and APM has been in existence for 10 years. APM asynchronous operations are started with a `Begin` method invocation. The result is obtained with an `End` method invocation. In Fig. 1b, `BeginGetResponse` is such a `Begin` method, and `EndGetResponse` is an `End` method.

Fig 2b shows the execution flow for the asynchronous code in Fig 1b. `BeginGetResponse` initiates an asynchronous `HTTP GET` request. The .NET framework starts the I/O operation in the background (in this case, sending the request to the remote web server). Control is returned to the calling method, the UI event handler in this case, which can then continue to do something else, thus it is responsive. When the server responds, the .NET framework will "call back" to the application to notify that the response is ready. `EndGetResponse` is then used in the callback code to retrieve the actual result of the operation.

Notice an important difference between the synchronous example in Fig. 1a/2a and the asynchronous, callback-based example in Fig. 1b/2b. In the synchronous example, the `Button_Click` method contains the UI update (setting the download result as contents of the text box). However, in the asynchronous example, the final callback contains an invocation of `Dispatcher.BeginInvoke(...)` to change context from the thread pool to the UI event thread and to post an update on the display (updating GUI elements from outside of the UI thread results in dataraces).

There are many other variations of the prevalent callback-based asynchronous programming. The .NET framework also supports a Task Asynchronous Programming (TAP) model, that uses lightweight `tasks`. The task represents the asynchronous operation and its future result. Other languages such as Java, Scala or Python also use the task-based concepts (sometimes called *futures*), though any long-running or blocking I/O operation needs to be encapsulated by the programmer into a task. For example, the Android app developer needs to encapsulate such tasks in `AsyncTask` (for short-running operations) or `IntentService` (for long-running operations).

B. Refactoring from callback-based asynchrony to `async/await` language constructs

So far, the dominant models for asynchronous APIs (e.g., Android, C# versions prior to 5.0, etc.) rely on programmers reasoning about callbacks. However, callbacks invert the control flow, are awkward, and obfuscate the intent of the original synchronous code [11], [12].

```
 1 void Button_Click(...) {
 2   var request = WebRequest.Create(”computer.org”);
 3   var response = request.GetResponse();
 4   var stream = response.GetResponseStream();
 5   textBox.Text = stream.ReadAsString();
 6 }
 7
 8
 9
10
11
12
13
14 .
```

(a) Original sync code

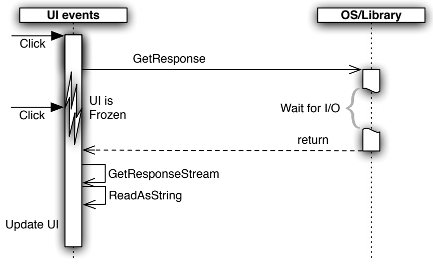
```
 1 void Button_Click(...) {
 2   var request = WebRequest.Create(”computer.org”);
 3   request.BeginGetResponse(Callback, request);
 4 }
 5
 6 void Callback(IAsyncResult aResult) {
 7   var request = (WebRequest)aResult.AsyncState;
 8   var response = request.EndGetResponse(aResult);
 9   var stream = response.getResponseStream();
10   var content = stream.ReadAsString();
11   Dispatcher.BeginInvoke(() => {
12     textBox.Text = content;
13   });
14 }
```

(b) callback APM

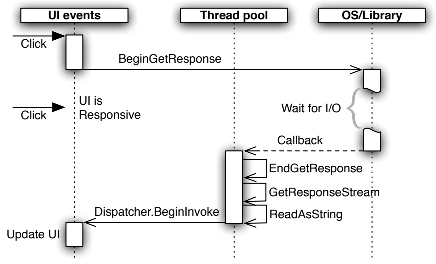
```
 1 async void Button_Click(...){
 2   var request = WebRequest.Create(”computer.org”);
 3   var response = await request.GetResponseAsync();
 4   var stream = response.GetResponseStream();
 5   textBox.Text = stream.ReadAsString();
 6 }
 7
 8
 9
10
11
12
13
14 .
```

(c) async/await

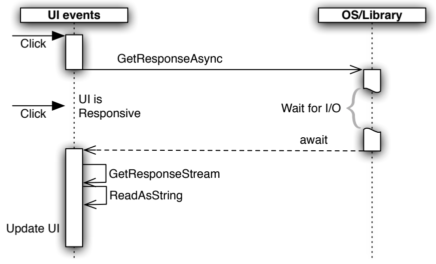
Fig. 1: Synchronous and two asynchronous versions of the same code for reading text from the web. Subfigure (a) shows the original synchronous code, (b) shows the asynchronous refactoring using callback-based constructs, (c) shows the `async/await` version.



(a) Execution of the synchronous code.



(b) Execution of the callback-based APM code



(c) Execution of the async/await-based code

Fig. 2: Runtime execution flow of the code in Fig. 1a–c. Notice that the time flows from top to bottom. In subfigure (a) the UI is frozen while waiting for the I/O operation, whereas in subfigures (b) and (c) the UI can process other events during the I/O operation.

Lets revisit our example in Fig 1b. Using the APM-style has two main drawbacks. First, the code that must be executed after the asynchronous operation is finished, must be passed explicitly to the `Begin` method invocation. Even more scaffolding is required: The `End` method must be called, and that usually requires the explicit passing and casting of an ‘async state’ `object` instance - see Fig 1b, lines 7–8. Second, even though the `Begin` method might be called from the UI event thread, the callback code is executed on a thread pool thread. To update the UI after completion of the asynchronous operation from the thread pool thread, an event must be sent to the UI event thread explicitly - see Fig 1b, line 11–13.

Recently, major programming languages (C# and Visual Basic [13], F# [11], and Scala [14]) introduced `async` constructs that resemble the straightforward coding style of traditional synchronous code. In the remainder of this section we will use the variant introduced by C# 5.0.

The model introduced by C# 5.0 is based on the `async` and `await` keywords. When a method has the `async` keyword modifier in its signature, the `await` keyword can be used to define pausing points. When a `Task` is awaited in an `await` expression, the current method is paused and control is returned to the caller. When the `await`’ed background operation is completed, the method is resumed from right after the `await` expression.

Figure 1c shows the `async/await`-based equivalent of Fig. 1b. The code following the `await` expression can be considered a continuation of the method, exactly like the callback that needs to be supplied explicitly when using APM or plain TAP. Figure 2c shows the execution flow for the `async/await`-based code in Fig. 1c.

There is one important difference between `async/await` continuations, and APM callback continuations: APM always executes the callback on a thread pool thread. In `async/await` continuations, the `await` keyword, by default, captures information about the thread in which it is executed. This captured context is used to schedule execution of the rest of the method in the same context as when the asynchronous operation was called. In our example, because the `await` keyword is encountered in the UI event thread, once the background operation is completed, the continuation of the rest of the method is scheduled back onto the UI event thread. This behavior allows the developer to write asynchronous code that resembles the original synchronous code (compare Fig. 1a and Fig. 1c).

With the advent of new `async/await` constructs, the asynchronous code looks deceptively similar to the synchronous code. While we applaud such engineering feats from the language designers and compiler writers, the app developer needs to be aware of the fundamental differences in the execution models, as illustrated in Fig. 2a, 2b, 2c.

III. Formative Studies

It is easy for academic research to become disconnected from the software practice and for researchers to build tools that do not solve immediate real-world problems. In order to avoid this, we started building a refactoring toolset for asynchrony which is grounded on empirical studies of how developers use, misuse, or underuse asynchrony.

We describe below two recent studies we conducted to obtain a deep understanding of the problems with asynchronous programming in Android and Windows Phone apps.

Study of Asynchrony in Android [7]. We conducted a formative study to understand how Android developers use asynchrony to improve responsiveness. We analyzed a corpus of the top 104 most popular Android apps in GitHub, comprising 1.34M SLOC, produced by 1139 developers. This formative study answers questions about use, misuse, and underuse of asynchrony. We found that 48% of the studied projects use asynchrony in hundreds of places, and half of these retrofitted asynchrony via manual refactoring. This shows that refactoring is in demand.

We also found that 251 places in 51 projects execute long-running operations in the UI event thread, and should have used asynchrony. We submitted refactoring patches for 6 apps. Developers of 4 apps replied and accepted 10 of our refactorings. This shows that refactorings are valuable.

Study of Asynchrony on Windows Phone [8]. For our Windows phone study, we analyzed 1378 open source Windows Phone (WP) apps, comprising 12M SLOC, produced by 3376 developers. This study helps us understand how programmers use asynchrony via the newly introduced language constructs `async/await` [13]. The next major version of Java also plans to support similar language constructs.

We found that in 76% of cases, developers use the old style of callback-based asynchronous idioms. However, Microsoft officially no longer recommends these asynchronous idioms [15]. Since refactoring callback-based idioms to new `async/await` idioms is non-trivial, there is a need for refactoring tools.

IV. Refactoring Obstacles

While asynchronous programming is key for improving responsiveness, it also presents several obstacles that developers must overcome. Based on our formative studies, in this section we present a list of top-10 questions that app developers must answer competently before they refactor synchronous into async code. Answering these questions wrongly, or not answering them at all, has severe consequences for (i) correctness [7] – resulting in data-races which are hard to debug, (ii) performance [8] – resulting in significant slowdowns or even deadlocks, (iii) maintainability [8], [16] – resulting in obsolete code, and (iv) usability – resulting in confusing interfaces.

Obstacle #1: Concurrency.

Below are some concurrency-related questions, related to (1) parallel execution, (2) data dependencies, (3) calling context, and (4) continuation:

1) What other code may run in parallel with the asynchronous code? Besides the main thread, are there other event handlers, or background threads spawned by the app, that could be running in parallel with the async code?
2) Are there dependencies between the code to be executed asynchronously and the other code that may run in parallel?
3) Is the code to be refactored called from within UI event thread or other thread?
4) After getting back the result from the async code, does the continuing code update the UI?

In our formative study of Android apps, we found that for 13 apps that contained manual refactorings, the asynchronous code accesses objects in a way which is not thread-safe. We discovered 77 data races on GUI widgets; while 53 races were already fixed by developers in later versions, we discovered and reported 24 new races along with the patch to fix them. The developers acknowledged and accepted our patch, showing that manual refactoring is error-prone.

Inspired by these developer needs, we released a refactoring tool, ASYNCHRONIZER, which helps the Android programmer to safely convert synchronous into asynchronous code. As is typical in the refactoring community, to ensure the safety of our automated refactorings, each refactoring is guarded by preconditions, i.e., certain criteria that the input code must satisfy before it can be safely refactored. Due to lack of space here, the interested reader can learn more about these preconditions in our research papers [7], [8], [16] and download the tools at [17].

Obstacle #2: Performance. Below are some performance questions, related to (1) non-blocking execution on the UI, (2) identifying other opportunities, and (3) vendor-specific guidelines:

1) After I encapsulate work within a task, does the code launch it asynchronously?
2) Is there any other long-running or I/O blocking code still in the UI thread?
3) Does my code follow the vendor-specific performance recommendations for the platform that I am targeting?

We found *misuse*s as real anti-patterns, which hurt performance and caused serious problems like deadlocks. In our Android case study, we found that in 4% of the cases, the code launches an asynchronous task and immediately blocks to wait for the task's result. Thus, the code appears to have async syntax, but runs synchronously instead of asynchronously. We found similar problems in our previous studies on concurrent libraries in C# [8], [18]. We found that 14% of methods that use (the expensive) `async/await` keywords do this unnecessarily: the awaited statement is the last one in an async method, thus the method will be paused unnecessarily since the async method will be awaited anyway at its call site. In the example below, the last `await` is unnecessary because the task returned by `SendPutRequestAsync` and then `ChangeTemperatureAsync` will be awaited anyway at the call site of `ChangeTemperatureAsync` (not shown here):

```
public async Task<?> ChangeTemperatureAsync(...) {
  ...
  return await SendPutRequestAsync(url, requestString);
}
```

In addition, in our WindowsPhone study, we found that 19% of methods do not follow an important good practice [19] that an async method should be awaitable unless it is the top level event handler. When an async method returns `void` instead of a `Task`, the code fires an async method which can not be awaited later on, thus wasting resources. We call this idiom "fire-and-forget".

Moreover, we found that 1 out of 5 apps misses opportunities in async methods to increase asynchrony.

We also found that developers (almost) always unnecessarily capture UI context, hurting performance. Recall the example from Fig. 1c where the remainder of the `async` method executes on the UI thread. As asynchronous GUI apps grow larger, there can be many small parts of `async` methods all using the UI event thread as their context. This can cause sluggishness as responsiveness suffers from "death by a thousand of paper cuts": the asynchronous code should run in parallel with the UI event thread instead of constantly badgering it with bits of work to do.

Such large numbers of misused `async/await` suggest a need for transformation tools that find and fix performance anti-patterns. Thus, we developed a tool, ASYNCFIXER, which detects and recommends fixes for several `async/await` anti-patterns. We encourage the reader to try it at: http://learnasync.net

Obstacle #3: Continuous, sometimes aggressive, API evolution. Like any useful API, the asynchronous APIs constantly evolve to support better constructs, hardware improvements that enable compiler optimizations, etc. Just because the developer has used asynchrony on her code, it does not mean that she will never change that async code again. The app developer must answer questions related to evolution of programming models:

1) Am I willing to learn a new programming model and change my already asynchronous code to support it?

We already presented in the previous section the refactoring from legacy callback-based async code into modern `async/await`. As another example, consider the evolution of async constructs inside of the Android platform. Android provides three major async constructs: `AsyncTask`, `IntentService`, and `AsyncTaskLoader`. `AsyncTask` is designed for encapsulating short-running tasks, while the other two are good choices for long-running tasks. However, as our most recent formative study [16] on a corpus of 611 Android apps shows, `AsyncTask` is the most widely used construct, dominating by a factor of 3x over the other two choices combined.

Using `AsyncTask` excessively for long-running tasks results in memory leaks, lost results, and wasted energy [16]. Thus, a developer might consider refactoring from an `AsyncTask` to `IntentService`. However, this refactoring is non-trivial due to drastic changes in communication with the GUI. This is a challenging problem because a developer needs to transform a shared-memory based communication (through access to the shared variables) into a distributed-style (through marshaling objects on special channels). Fortunately, this is a problem that is in the realm of semi-automated refactoring tools.

Inspired by these obstacles, we released two refactoring tools that modernize existing async code: (i) ASYNCDROID enables the Android programmer to convert `AsyncTask` to `IntentService`; (ii) ASYNCIFIER enables the .NET programmer to upgrade APM call-back style code into modern `async/await`. Both are available at [17].

Obstacle #4: Changing the UI paradigm. As a result of retrofitting asynchrony, the app developer might have to change the UI paradigm as well. Consider again the motivating example from Fig. 1. Using the UI (not shown here) for the synchronous model of Fig. 1a, a user can only press the button once, then has to wait for the download to finish before executing other UI actions. With the asynchronous models of Fig. 1b–c, the user can click the download button multiple times consecutively. Thus, multiple requests for downloading the same page might be in progress at the same time. This can frustrate the user and/or lead to inefficient use of resources. The app developer must answer questions related to (1) UI workflow, and (2) UI decomposition:

1) If an async operation is already in progress, should the triggering UI widget be disabled?
2) How can I group my UI widgets to balance responsiveness (making the user feel in control) and efficient use of resources (to avoid wasted subsequent requests that are overriding each others' result)?

Answering such questions might require the developer to rethink the UI workflow model and layout. For example, a developer might disable a UI widget while an async operation is in progress, or show progress report during long-running operations, etc.

V. PRACTICAL IMPACT OF OUR REFACTORING TOOLSET

A. *Online Educational Resources*

In our formative studies, we have seen extensive underuse and misuse of asynchronous constructs. This raises the question: Why is the misuse so extensive? Are developers unaware of the risks or performance characteristics of `async/await`?

To significantly improve education, we developed a portal, http://learnasync.net with educational resources for the .NET programmer who wants to learn about async constructs.

Developers learn a new programming construct through both positive and negative examples. Thus, on our portal, we provide thousands of real-world examples of all asynchronous idioms. Because developers might need to inspect the whole source file or project to understand the example, our portal links to highlighted source files on GitHub, so developers see the async code in its surrounding context.

So far, our portal has received more than 36,000 visitors within the 2 years since we launched it.

B. *Transformation tools*

Android tools. Our refactoring tool, ASYNCHRONIZER, enables app developers to extract long-running operations into `AsyncTask`, the primary callback-based async construct in Android.

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 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. Developers of 5 apps 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. Developers of 4 apps replied and accepted 10 refactorings. This shows that ASYNCHRONIZER is valuable.

The tool and experimental subjects are available for download at: http://refactoring.info/tools/asynchronizer

.NET tools. We developed both refactoring tools (ASYNCIFIER), and tools to find and fix misuses of async constructs (ASYNCFIXER). In our paper [8] we show that our tools are highly applicable and efficient. Developers find our transformations useful. Using ASYNCIFIER, we applied and reported refactorings in 10 apps. 9 replied and accepted each one of our 28 refactorings. The developer of PHONEGUITARTAB said that he had "been thinking about replacing all asynchronous calls [with] new async/await style code". This illustrates the demand for tool support for refactoring.

Using ASYNCFIXER, we found and reported misuses in 19 apps. 19 replied and accepted each of our 286 patches. This shows that developers deeply care about these problems. The developer of SOFTBUILDDATA experienced performance improvements after applying our patch: "[...] response time has been improved to 28 milliseconds from 49 milliseconds."

A subset of our ASYNCIFIXER will ship with the official release of Visual Studio at the end of 2015. Developers can find demos and download our tools at: http://learnasync.net

C. *From individual to ecosystem outreach*

There are several ways for researchers to move their research into practice: to connect with individual developers and companies, or to connect with whole communities of developers. We call the former the "downstream" approach, because the researcher integrates the research at the end of an already existing pipeline of tools that developers already use. We call the latter the "upstream" approach, because the researcher packages the research into an ecosystem of tools that gets supplied to a large community of developers. We have been pursuing both outreach approaches.

On the individual outreach, we have been submitting refactoring patches to hundreds of open-source projects. Moreover, we recently replicated our open-source study [8] of asynchronous constructs on the code-base of an industrial partner from the Pacific NorthWest. At the end of this experience, the industrial partner had a clear sense of how their proprietary codebase compares to the open-source projects in terms of density of usage, misusage, and under-usage of async constructs. This was a clearly a win-win situation: we expanded our research on case studies that would otherwise be unavailable to us, while the company gained a deeper understanding of their async code practices and a list of actionable items. We are proactively looking for new partnerships.

On the ecosystem approach, we have been already working with IDE developers (Visual Studio, Eclipse, NetBeans) by contributing our research as plugins that already ship with the official IDE version. We will continue to explore this even more. We are currently working with Google to deploy our refactoring and transformation tools as analyzers for the Shipshape [20] static analysis platform. The vision is for Shipshape to become a widely-used platform. Any app developer that wants to check code quality, for example before submitting an app to the app store, would run Shipshape on her code base. Shipshape allows custom analyzers, such as our async analysis and transformations, to plug in through a common interface. Shipshape generates analysis results along with transformation patches that developers can accept on their code. We expect that by contributing new async analyzers to ShipShape, millions of app developers would benefit by being able to execute our analysis and transformations on their code.

While the ecosystem outreach approach provides a massive impact opportunity, the individual outreach has its own benefits. Among others, it provides unique points of interaction with developers, early feedback from users, deep insights into problems that industry faces, and the ability to replicate open-source experiments on industrial code bases. Thus, we are always interested in industrial partners that want to engage with us.

VI. RELATED WORK

Empirical studies [1], [21] of performance bug patterns in Android apps revealed that lack of responsiveness is the main culprit. Testing researchers [22]–[25] have used machine learning, concolic, and random testing to generate test inputs for mobile apps. However, our work is complementary: we explore how programmers can use refactoring to eliminate the performance issues that are detected by testing.

While refactoring has been traditionally associated with improving the design of code, the refactoring community has been taking a similar approach to improve other non-functional requirements, for example performance. In 2009 we published the first research paper [26] that opened the field of interactive refactorings for parallelism, and continued to publish extensively on this topic, see our summary paper [27] that also describes the related work in this field, afferent tools are available online [17].

While our formative studies [7], [8], [16] identified misuse and underuse of refactoring in the specific context of responsiveness in mobile apps, other researchers have described such problems in general software: Kim et al. [28] present a field study at Microsoft on refactoring challenges and benefits, Murphy-Hill et al. [29] present the state of the practice in refactoring usage, others [30], [31] studied use/disuse/misuse of refactoring in general.

VII. CONCLUSION

With their rich array of sensors, camera, and GPS, all integrated in a convenient form factor, mobile devices can offer new and exciting applications and services that were not possible before on consumer devices. But these can be harnessed only if mobile applications leverage asynchrony to ensure responsive behavior.

This paper gives a flavor of the kinds of refactorings that app developers must perform on their code base in order to improve responsiveness. We presented the benefits, but also the obstacles of asynchrony, along with our ongoing refactoring tools to retrofit asynchrony, as well as to migrate legacy async code to modern constructs. We also present transformation tools that find and fix misuses of async constructs.

Moreover, we hope that our educational resources show developers the potential as well as the pitfalls of using async constructs in their code. We are constantly looking for industrial partners that we can help to discover their async practices and refactor their code to improve responsiveness.

Acknowledgment

The author would like to thank Yu Lin, Semih Okur, Michael Hilton, Sruti Srinivasa, Lily Mast, and the anonymous reviewers for insightful comments on earlier drafts of this paper. 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] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *International Conference on Software Engineering (ICSE)*, 2014, pp. 1013–1024.

[2] Gartner, July'14, http://www.gartner.com/newsroom/id/2153215.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[4] B. Opdyke and R. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *SOOPPA'90: Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[5] W. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, University of Washington, 1991.

[6] B. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[7] Y. Lin, C. Radoi, and D. Dig, "Retrofitting Concurrency for Android Applications through Refactoring," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2014, pp. 341–352.

[8] S. Okur, D. L. Hartveld, D. Dig, and A. van Deursen, "A study and toolkit for asynchronous programming in C#," in *International Conference on Software Engineering (ICSE)*, 2014, pp. 1117–1127.

[9] "Android Best Practices for Performance," http://developer.android.com/training/articles/perf-anr.html.

[10] "StrictMode – Android framework tool for detecting long-running operations in the UI thread," http://developer.android.com/reference/android/os/StrictMode.html.

[11] D. Syme, T. Petricek, and D. Lomov, "The F# asynchronous programming model," in *International Conference on Practical Aspects of Declarative Languages (PADL)*, 2011, pp. 175–189.

[12] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini, "An Empirical Study on Program Comprehension with Reactive Programming," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2014, p. To Appear.

[13] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen, "Pause n Play: Formalizing Asynchronous CSharp," in *European conference on Object-Oriented Programming (ECOOP)*, 2012, pp. 233–257.

[14] S. Async, July'14, http://docs.scala-lang.org/sips/pending/async.html.

[15] M. A. P. Patterns, July'14, http://msdn.microsoft.com/en-us/library/jj152938.aspx.

[16] Y. Lin, S. Okur, and D. Dig, "Study and Refactoring of Android Asynchronous Programming," Oregon State University, School of Electrical Engineering and Computer Science, Tech. Rep. http://hdl.handle.net/1957/56106, 2015.

[17] "Homepage for our refactoring for parallelism toolset," http://refactoring.info/tools.

[18] S. Okur and D. Dig, "How do developers use parallel libraries?" in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 54–65.

[19] B. P. in Asynchronous Programming, July'14, http://msdn.microsoft.com/en-us/magazine/jj991977.aspx.

[20] "ShipShape analysis framework," https://github.com/google/shipshape.

[21] 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*, ser. MOBS '13, 2013, pp. 1–6.

[22] 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*, ser. OOPSLA '13, 2013, pp. 623–640.

[23] 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*, ser. ISSTA '13, 2013, pp. 67–77.

[24] 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*, ser. FSE '12, 2012, pp. 1–11.

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

[26] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 397–407.

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

[28] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2012, p. 50.

[29] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *ICSE*, 2009, pp. 287–297.

[30] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *International Conference on Software Engineering*, 2012, pp. 233–243.

[31] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *ECOOP: European Conference on Object Oriented Programmming*, 2013, pp. 552–576.

PLACE PHOTO HERE

Danny Dig Biography text here.