Refactoring for Asynchronous Execution on Mobile Devices

Danny Dig, Oregon State University

// Refactoring that changes synchronous code to asynchronous code can improve mobile apps' responsiveness. However, using asynchrony presents obstacles. Researchers have developed educational resources and tools that can help. //

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

Asynchrony is especially valuable for applications that access the UI thread. Today's UI frameworks are usually designed around a single UI event thread: every operation that modifies UI state executes atomically as an event on that thread.[1] The UI freezes when it can't respond to input or can't be redrawn; because it seems nonresponsive, users might become frustrated. It's 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 touchscreens, accelerometers, microphones, and other sensors on smartphones. Similarly, modern Web applications extensively use asynchrony, through AJAX (asynchronous JavaScript and XML) requests, and use asynchronous code loading to reduce the perceived page load time.

This article focuses on mobile apps because they provide many exemplars of asynchronous programming, given that responsiveness is critical. They can easily be unresponsive because mobile devices have limited resources and high latency (excessive network accesses).[2] With the immediacy of touch-based UIs, even small hiccups in responsiveness are more obvious and jarring than with a mouse or keyboard. Some sluggishness might motivate users to uninstall an app and submit negative comments in the app store. Here, I show how refactoring synchronous code into asynchronous code can improve mobile apps' responsiveness and avoid these problems.

The Problems with Asynchrony

When programmers work with asynchrony, they encounter two key problems.

The Lack of Methods and Tools

The first problem is the lack of methods and tools for converting synchronous code to asynchronous code. In formative studies of hundreds of open source apps, my colleagues and I found that half of the asynchrony usages weren't introduced from scratch but had been converted from preexisting synchronous code. This step is a source-to-source transformation that must preserve the application's current functionality (while improving some nonfunctional property such as responsiveness). So, it's a refactoring.[3]

This refactoring requires complex code transformations that invert the control flow and introduce callbacks to notify the caller when an asynchronous operation finishes. Also, it requires reasoning about asynchronous operations' noninterference with the main thread of execution; otherwise, the asynchrony can lead to nondeterministic data races.

Currently, developers solve such problems and perform such nontrivial transformations manually. However, interactive refactoring tools can help.

The Lack of Knowledge

The second problem is the lack of clear knowledge of how to convert synchronous code into asynchronous code. There's extensive programmer documentation (for example, Android's *Best Practices for Performance*[1]) and tools that detect blocking-I/O operations (for example, StrictMode for Android[4]). But they focus on designing asynchronous programs from scratch. So, many programs suffer from poor responsiveness.

A recent paper found that 75 percent of performance bugs in Android apps arise because of missed opportunities to introduce asynchrony in UI code.[2] Our research discovered hundreds of places in UI code that should have used asynchrony.[5,6] This shows the need for educational resources on how to retrofit asynchrony.

Dealing with the Problems

My colleagues and I have addressed some of these challenges by releasing educational resources on our online portal, http://learnasync.net. We've also developed tools that automatically refactor synchronous code into asynchronous code. We discuss our portal and tools in more detail later. You can learn more about the tools and download them at http://refactoring.info/tools.

Refactorings for Asynchrony

Examples from GUI programming on mobile devices illustrate refactoring flavors and the refactorings my colleagues and I have developed. Although the code examples use C# and .NET APIs, similar constructs exist or are planned for Java and Android.

Most GUI frameworks use an event-driven model. Events in mobile apps include life-cycle events (for example, GUI screen creation), user actions (for example, button click and menu selection), and sensor inputs (for example, GPS and screen orientation change), and so on. Developers define event handlers to respond to these events.

To handle app events, the operating system creates a main thread—the UI event thread. This thread dispatches UI events to appropriate widgets or life-cycle events to screen pages. It puts events into an event queue, dequeues events, and executes the corresponding event handlers.

When a GUI event handler executes a synchronous long-running CPU-bound or blocking-I/O operation, the UI freezes because the UI event thread can't respond to events. Figure 1a shows a handler that responds to a button click by downloading and displaying the IEEE Computer Society homepage. The main thread invokes a blocking, potentially long-running operation, getResponse, which downloads the webpage, and the UI becomes unresponsive during the download. Figure 2a shows this code's execution flow, highlighting when the UI freezes.

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

Next, I present the two prevalent asynchronous-programming styles using examples from .NET:

- the framework-based, callback-based style, which uses the Asynchronous Programming Model (APM) from .NET 1.0, and
- the new pause-and-play style based on the asynchronous language constructs async and await (called async/await in the rest of this article) from the most recent version of .NET

Our refactoring tools target both styles.

Refactoring to Callback-Based Asynchrony

Figure 1b shows the refactored, asynchronous version of the code in Figure 1a. A Begin method (BeginGetResponse) invocation starts APM asynchronous operations; an End method (EndGetResponse) invocation obtains the results.

Figure 2b shows the execution flow for Figure 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 webserver). Control returns to the calling method, in this case the UI event handler, which can then continue to do something else; thus, it is responsive. When the server responds, the .NET

```
void Button_Click(...) {
  var request = WebRequest.Create("computer.org");
  var response = request.GetResponse();
  var stream = response.GetResponseStream();
  textBox.Text = stream.ReadAsString();
}








.
```
(a)

```
void Button_Click(...) {
  var request = WebRequest.Create("computer.org");
  request.BeginGetResponse(Callback, request);
}

void Callback(IAsyncResult aResult) {
  var request = (WebRequest)aResult.AsyncState;
  var response = request.EndGetResponse(aResult);
  var stream = response.getResponseStream();
  var content = stream.ReadAsString();
  Dispatcher.BeginInvoke(() => {
    textBox.Text = content;
  });
}
```
(b)

```
async void Button_Click(...){
  var request = WebRequest.Create("computer.org");
  var response = await request.GetResponseAsync();
  var stream = response.GetResponseStream();
  textBox.Text = stream.ReadAsString();
}








.
```
(c)

FIGURE 1. Three versions of the same code for reading text from the Web. (a) The original synchronous code. (b) The asynchronous refactoring using callback-based constructs. (c) The **async/await** version. In the asynchronous versions, the UI can process other events during the I/O operation.

framework "calls back" to the application to notify it that the response is ready. The callback code then uses **EndGetResponse** to retrieve the operation's results.

An important difference exists between the synchronous example in Figures 1a and 2a and the asynchronous, callback-based example in Figures 1b and 2b. In the synchronous example, the **Button_Click** method contains the UI update (setting the download result as the text box's contents). In the asynchronous example, the final callback contains an invocation of **Dispatcher.BeginInvoke(...)** to change the context from the thread pool to the UI event thread and to update the display. (Updating GUI elements from outside the UI thread causes data races.)

Callback-based asynchronous programming has many variations. Several languages (such as Java, C#, and Scala) support lightweight *tasks*. A task can represent an asynchronous operation and its future result. Developers still must encapsulate any long-running operations or blocking I/O as tasks. For example, Android developers must encapsulate asynchronous operations in **AsyncTask** or **IntentService**. (We discuss these constructs in more detail later.) .NET developers can use the Task Asynchronous Programming (TAP) model.

Refactoring to **async/await**

So far, the dominant models for asynchronous APIs (for example, Android and C# versions before 5.0) rely on programmers reasoning about callbacks. However, callbacks invert the control flow, are awkward, and obfuscate the original synchronous code's intent.[7,8]

Let's revisit the example in Figure 1b. Using the APM style has two main drawbacks. First, the code

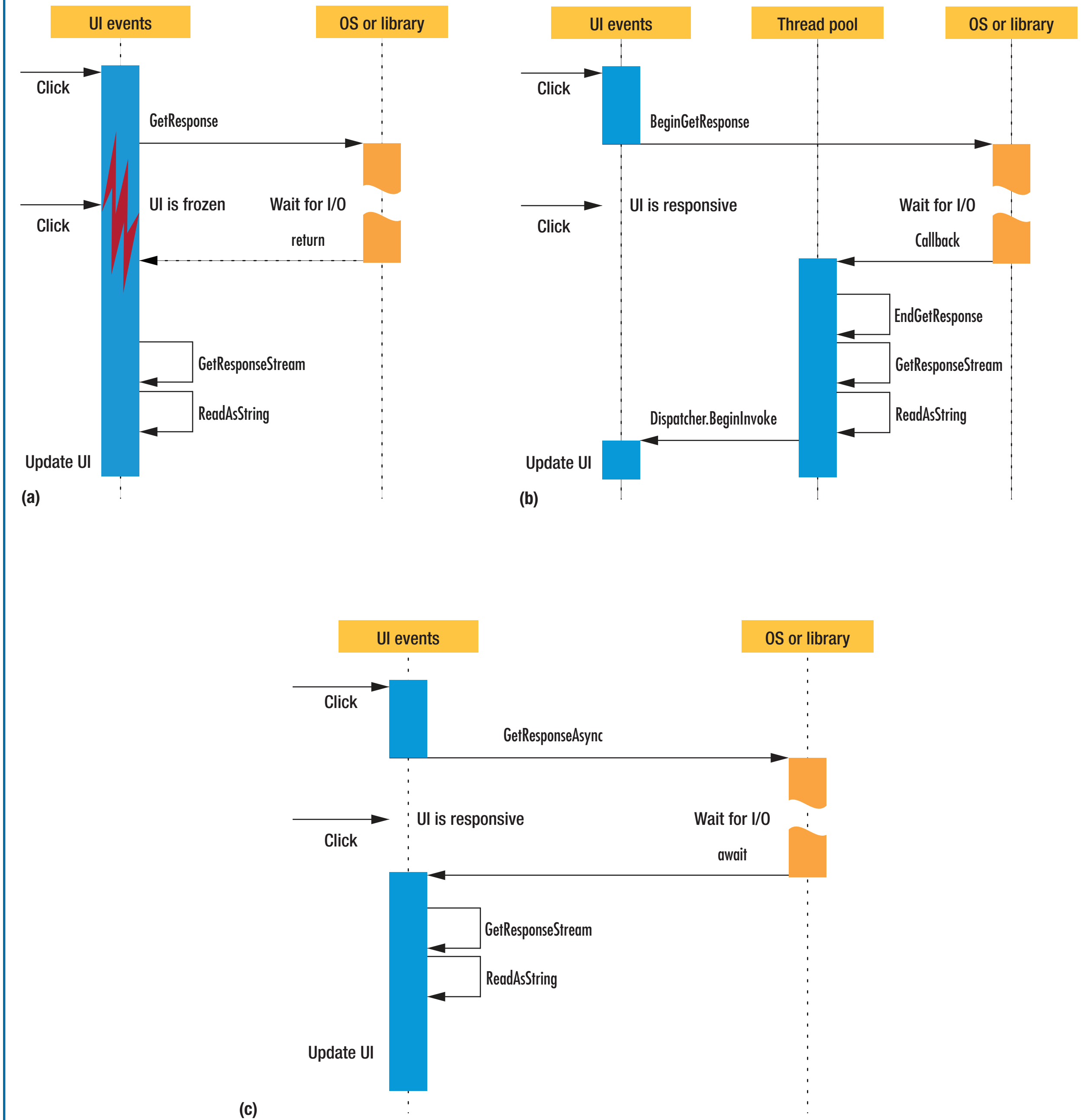

FIGURE 2. Run-time execution flow of the code in Figure 1. The time flows from top to bottom. (a) The execution flow of the code in Figure 1a. The UI freezes while waiting for the I/O operation. (b) The execution flow for the asynchronous code in Figure 1b. (c) The execution flow for the async/await-based code in Figure 1c. In Figures 2b and 2c, the UI can process other events during the I/O operation.

that must execute after the asynchronous operation finishes must be passed explicitly to the **Begin** method invocation. Even more scaffolding is required. The **End** method must be called, which usually requires the explicit passing and casting of an **AsyncState object** instance (see Figure 1b, lines 7 and 8).

Second, even though the **Begin** method might be called from the UI event thread, the callback code executes on a thread pool thread. To update the UI after the asynchronous operation from that thread completes, an event must be sent to the UI event thread explicitly (see Figure 1b, lines 11 to 13).

Recently, major programming languages (C# and Visual Basic,[9] F#,[7] and Scala[10]) introduced asynchronous constructs that resemble the straightforward coding style of traditional synchronous code. In the rest of this section, I use the variant introduced by C# 5.0.

This variant 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 pauses and control is returned to the caller. When the **await**ed background operation finishes, the method resumes from right after the **await** expression.

Figure 1c shows the **async/await**-based equivalent of Figure 1b. The code following the **await** expression can be considered a continuation of the method, exactly like the callback that must be supplied explicitly when using APM. Figure 2c shows the execution flow for Figure 1c.

An important difference exists between APM callback continuations and **async/await** 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 executes. 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 my example, because the **await** keyword is encountered in the UI event thread, once the background operation finishes, the continuation of the rest of the method is scheduled back onto the UI event thread. This behavior lets developers write asynchronous code that resembles the original synchronous code (compare Figures 1a and 1c).

With the advent of **async/await** constructs, the asynchronous code looks deceptively like the synchronous code. Although I applaud such engineering feats from the language designers and compiler writers, app developers must be aware of the execution models' fundamental differences (see Figure 2).

We found misuses as real antipatterns, which hurt performance and caused serious problems such as deadlocks.

Formative Studies

It's easy for academic research to become disconnected from the software practice and for researchers to build tools that don't solve immediate real-world problems. To avoid this, our refactoring tools are grounded on empirical studies of how developers use, misuse, or underuse asynchrony.

Specifically, to obtain a deep understanding of the problems with asynchronous programming in Android and Windows Phone apps, my colleagues and I conducted the following two studies.

Asynchrony in Android

We conducted this study to understand how Android developers use asynchrony to improve responsiveness.[5] We analyzed the 104 most popular Android apps in GitHub, comprising 1.34M SLOC, produced by 1,139 developers. We found that 48 percent of those projects used asynchrony in hundreds of places, and half of them retrofitted asynchrony through manual refactoring.

We also found that 251 places in 51 projects executed long-running operations in the UI event thread and should have used asynchrony. We submitted refactoring patches for six apps. Developers of four apps replied and accepted 10 of our refactorings.

Asynchrony on Windows Phones

This study helped us understand how programmers use asynchrony through **async/await**.[9] (The next major version of Java plans to support similar constructs.) We analyzed 1,378 open source Windows Phone apps, comprising 12M SLOC, produced by 3,376 developers.[6]

In 76 percent of cases, developers used callback-based asynchronous idioms. However, Microsoft officially no longer recommends these idioms.[11] Because refactoring these idioms to async/await idioms is nontrivial, the need exists for refactoring tools.

Refactoring Obstacles

On the basis of the formative studies, here I present the top 10 questions app developers must answer competently before refactoring synchronous code into asynchronous code. Answering these questions wrongly or not at all has severe consequences for

- correctness,[5] resulting in hard-to-debug data races;
- performance,[6] resulting in significant slowdowns or even deadlocks;
- maintainability,[6,12] resulting in obsolete code; and
- usability, resulting in confusing interfaces.

These questions fall into the following four categories related to the obstacles to asynchronous programming.

Obstacle 1: Concurrency

Four questions are concurrency related:

- 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 run in parallel?
- Do dependencies exist between the code to be executed asynchronously and the other code that may run in parallel?
- Is the code to be refactored called from within the UI event thread or another thread?
- After getting back the result from the asynchronous code, does the continuing code update the UI?

Our study of Android apps found that for 13 apps containing manual refactorings, the asynchronous code accessed objects in a way that wasn't thread-safe. We discovered 77 data races on GUI widgets. Developers had already fixed 53 of those races in later versions, and we discovered and reported 24 new races along with the patch to fix them. The developers acknowledged and accepted our patch.

Inspired by these developer needs, we released *Asynchronizer*, a tool that helps Android programmers safely refactor synchronous code into asynchronous code. As is typical in the refactoring community, to ensure our automated refactorings' safety, each refactoring is guarded by preconditions—criteria the input code must satisfy before it can be safely refactored.

Obstacle 2: Performance

Three questions are performance related:

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

We found misuses as real antipatterns, which hurt performance and caused serious problems such as deadlocks. Our Android app study found that in 4 percent of the cases, the code launched an asynchronous task and immediately blocked to wait for the task's result. So, the code appeared to have asynchronous syntax but ran synchronously instead of asynchronously.

Developers almost always unnecessarily captured UI context, hurting performance.

Our previous studies on concurrent libraries in C# discovered similar problems.[6,13] We found that 14 percent of methods that used (the expensive) async/await keywords did this unnecessarily. The awaited statement was the last one in an async method, so the method would pause unnecessarily because it would be awaited anyway at its call site.

In the following example, 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<?> ChangeTempera-
tureAsync(...) {
...
return await SendPutRequestAsync(url,
requestString);
}
```

In addition, our Windows Phone study discovered that 19 percent of methods didn't follow the important practice that an async method should be awaitable unless it's the top-level event handler.[14] When an async method returned void instead of a Task, the code fired an async method that couldn't be awaited, thus wasting resources. We call this idiom "fire and forget."

Moreover, we found that one of five apps missed opportunities in async methods to increase asynchrony.

We also found that developers almost always unnecessarily captured UI context, hurting performance. Recall the example from Figure 1c where the remainder of the async method executes on the UI thread. As asynchronous GUI apps grow larger, many small parts of async methods might use the UI event thread as their context. This can cause sluggishness as responsiveness suffers "death by a thousand 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.

So much misuse of async/await suggests the need for transformation tools that find and fix performance antipatterns. So, we developed *AsyncFixer*, a tool that detects several async/await antipatterns and recommends fixes for them.

Our portal provides thousands of real-world examples of all asynchronous idioms.

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, and so on. Just because developers have used asynchrony on their code doesn't mean they won't ever change that asynchronous code. App developers must answer questions related to programming-model evolution—for example,

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

Consider the evolution of asynchronous constructs in the Android platform. Android provides three major asynchronous constructs: AsyncTask, IntentService, and AsyncTaskLoader. AsyncTask is for encapsulating short-running tasks; the other two are good for long-running tasks. However, as our most recent study on a corpus of 611 Android apps showed, AsyncTask is the most widely used construct, dominating by a factor of 3× over the other two choices combined.[12]

Using AsyncTask excessively for long-running tasks causes memory leaks, lost results, and wasted energy.[12] So, developers might consider refactoring AsyncTask into IntentService. However, this refactoring is nontrivial owing to drastic changes in communication with the GUI. This is a challenge because developers must transform shared-memory-based communication (through access to the shared variables) into distributed communication (through marshaling objects on special channels). Fortunately, semiautomated refactoring tools can handle this challenge.

Inspired by these obstacles, we released two tools that modernize existing asynchronous code. *AsyncDroid* lets Android programmers convert AsyncTask to IntentService. *Asyncifier* lets .NET programmers upgrade APM callback-style code to modern async/await.

Obstacle 4: Changing the UI paradigm

As a result of retrofitting asynchrony, app developers might have to change the UI paradigm too. Consider again the example from Figure 1. With the UI for the synchronous model of Figure 1a, users can only press the button once and then must wait for the download to finish before executing other UI actions. With the asynchronous models of Figures 1b and 1c, users can click the download button multiple times consecutively. So, multiple requests for downloading the same page might be in progress simultaneously. This can frustrate users and lead to inefficient use of resources.

App developers must answer questions such as these:

- If an async operation is already in progress, should the triggering UI widget be disabled?
- 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 override each other's results)?

Answering such questions might require developers to rethink the UI workflow model and layout. For example, developers might disable a UI widget while an asynchronous operation is in progress or have the app display a progress report during long-running operations.

Our Research's Practical Impact

Our research has had a practical impact in the following areas.

Our Online Portal

In our studies, we've seen extensive underuse and misuse of asynchronous constructs. Why is the misuse so extensive? Are developers unaware of the risks or performance characteristics of async/await?

To significantly improve education, our online portal provides resources for .NET programmers who want to learn about asynchronous constructs. Developers learn new programming constructs through both positive and negative examples. So, our portal provides thousands of real-world examples of all asynchronous idioms. Because developers might need to inspect the whole source file or project to understand an example, our portal links to highlighted source files on GitHub so that developers see the asynchronous code in its context.

Since we launched the portal two years ago, it has had more than 36,000 visitors.

Our Refactoring Tools

Here, we look at three of our tools: Asynchronizer, for Android, and Asyncifier and AsyncFixer, for .NET.

Asynchronizer. This tool lets app developers extract long-running operations into AsyncTask. We used it to refactor 135 places in 19 open source Android projects. We evaluated its usefulness from five angles. First, because 95 percent of the cases met refactoring preconditions, refactoring was highly applicable. Second, in 99 percent of cases, Asynchronizer applied changes similar to those that open source developers applied manually; thus, our transformation was accurate. Third, Asynchronizer changed 2,394 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 five apps replied and confirmed 62 races. Fifth, we submitted patches for 58 refactorings in six apps. Developers of four apps replied and accepted 10 refactorings.

Asyncifier and AsyncFixer. Our previous research showed that Asyncifier and AsyncFixer are highly applicable and efficient.[6] Developers find our transformations useful.

Using Asyncifier, we applied and reported refactorings in 10 apps. Nine of the developers replied and accepted our 28 refactorings. PhoneGuitarTab's developer said that he had "been thinking about replacing all asynchronous calls [with] new async/await-style code."

Using AsyncFixer, we found and reported misuses in 19 apps. All the developers replied and accepted our 286 patches. After applying our patch, Softbuilddata's developer experienced performance improvements: "Response time has been improved to 28 milliseconds from 49 milliseconds."

A subset of AsyncFixer will ship with the official release of Visual Studio at the end of 2015.

Individual and Ecosystem Outreach

Researchers can move their research into practice by connecting with individual developers and companies or with whole communities of developers. We call the former the "downstream" approach because the researcher integrates the research at the end of an 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've been pursuing both approaches.

The downstream approach. This approach 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.

ABOUT THE AUTHOR

DANNY DIG is an assistant professor in Oregon State University's Electrical Engineering and Computer Science department and an adjunct assistant professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests are automated refactoring, interactive program analysis and transformation, and design and architectural patterns. Dig received a PhD in computer science from the University of Illinois at Urbana-Champaign. His dissertation on API refactoring won the University of Illinois David Kuck Outstanding PhD Thesis Award and First Prize at the ACM Student Research Competition Grand Finals. Contact him at digd@eecs.oregonstate.edu.

RELATED WORK IN PROGRAM PERFORMANCE AND REFACTORING

Empirical studies of performance bug patterns in Android apps revealed that lack of responsiveness was the main culprit.[1,2] Testing researchers have used machine learning,[3] concolic testing,[4] and random testing[5,6] to generate test inputs for mobile apps. My research is complementary; I explore how programmers can use refactoring to eliminate performance issues detected by testing.

Although refactoring has usually been associated with improving code design, the refactoring community has been taking a similar approach to improve other nonfunctional requirements—for example, performance. In 2009 my colleagues and I published the research paper that opened the field of interactive refactorings for parallelism.[7] We've continued to publish extensively on this topic, including a summary paper that describes related work in this field.[8]

Whereas our formative studies identified misuse and underuse of refactoring in the context of responsiveness in mobile apps,[9–11] other researchers have described such problems in general software. Miryung Kim and her colleagues presented a field study at Microsoft on refactoring challenges and benefits.[12] Emerson Murphy-Hill and his colleagues reported on the state of the practice in refactoring usage.[13] Other researchers have studied the general use, disuse, and misuse of refactoring.[14,15]

References

1. Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1013–1024.
2. S. Yang, D. Yan, and A. Rountev, "Testing for Poor Responsiveness in Android Applications," *Proc. 1st Int'l Workshop Eng. of Mobile-Enabled Systems* (MOBS 13), 2013, pp. 1–6.
3. W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," *Proc. 28th ACM SIGPLAN Int'l Conf. Object Oriented Programming Systems Languages & Applications* (OOPSLA 13), 2013, pp. 623–640.
4. S. Anand et al., "Automated Concolic Testing of Smartphone Apps," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 59.
5. C.S. Jensen, M.R. Prasad, and A. Møller, "Automated Testing with Targeted Event Sequence Generation," *Proc. 13th Int'l Symp. Software Testing and Analysis* (ISSTA 13), 2013, pp. 67–77.
6. C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," *Proc. 6th Int'l Workshop Automation of Software Test* (AST 11), 2011, pp. 77–83.
7. D. Dig, J. Marrero, and M.D. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 397–407.
8. D. Dig, "A Refactoring Approach to Parallelism," *IEEE Software*, vol. 28, no. 1, 2011, pp. 17–22.
9. Y. Lin, C. Radoi, and D. Dig, "Retrofitting Concurrency for Android Applications through Refactoring," *Proc. ACM SIGSOFT 22nd Int'l Symp. Foundations of Software Eng.* (FSE 14), 2014, pp. 341–352.
10. S. Okur et al., "A Study and Toolkit for Asynchronous Programming in C#," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1117–1127.
11. Y. Lin, S. Okur, and D. Dig, "Study and Refactoring of Android Asynchronous Programming," tech. report, School of Electrical Eng. and Computer Science, Oregon State Univ., 2015; http://hdl.handle.net/1957/56106.
12. M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 50.
13. E.R. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 287–297.
14. M. Vakilian et al., "Use, Disuse, and Misuse of Automated Refactorings," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 233–243.
15. S. Negara et al., "A Comparative Study of Manual and Automated Refactorings," *Proc. 27th European Conf. Object-Oriented Programming* (ECOOP 13), 2013, pp. 552–576.

As part of this approach, we're submitting refactoring patches to hundreds of open source projects. Moreover, we recently replicated our open source study of asynchronous constructs[6] on the proprietary code base of an industrial partner from the Pacific Northwest. The partner obtained a clear sense of how its code base compared to the open source projects in terms of the use, misuse, and underuse of `async` constructs.

This was clearly a win–win situation. We expanded our research on case studies that otherwise would have been unavailable to us; the company gained both a deeper understanding of their asynchronous-code practices and a list of actionable items.

The upstream approach. This approach can have a massive impact. As part of this approach, we're working with IDE developers (Visual Studio, Eclipse, and NetBeans) by contributing our research as plug-ins that ship with the official IDE version. We'll explore this even more.

We're also working with Google to deploy our refactoring and transformation tools as analyzers for the Shipshape static-analysis platform (https://github.com/google/shipshape). The vision is for Shipshape to become widely used. Developers who want to check code quality—for example, before submitting an app to Google Play—would run Shipshape on their code base. Shipshape lets custom analyzers, such as our `async` analysis and transformations, 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, we'll enable millions of app developers to execute our analyses and transformations on their code.

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

We hope our educational resources show developers asynchronous constructs' potential and pitfalls. We're constantly looking for industrial partners we can help to discover their asynchronous practices and refactor their code to improve responsiveness.

For a look at other research on program performance and refactoring, see the sidebar.

Acknowledgments
I thank Yu Lin, Semih Okur, Michael Hilton, Sruti Srinivasa, Lily Mast, and the anonymous reviewers for insightful comments on earlier drafts of this article. US National Science Foundation grants CCF-1439957 and CCF-1442157, a Microsoft Software Engineering Innovation Foundation award, and an Intel gift grant partly funded this research.

References

1. *Best Practices for Performance*, Android Open Source Project; http://developer.android.com/training/best-performance.html.
2. Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1013–1024.
3. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. "StrictMode," Android Open Source Project; http://developer.android.com/reference/android/os/StrictMode.html.
5. Y. Lin, C. Radoi, and D. Dig, "Retrofitting Concurrency for Android Applications through Refactoring," *Proc. 2014 ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (FSE 14), 2014, pp. 341–352.
6. S. Okur et al., "A Study and Toolkit for Asynchronous Programming in C#," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1117–1127.
7. D. Syme, T. Petricek, and D. Lomov, "The F# Asynchronous Programming Model," *Proc. 2011 Int'l Conf. Practical Aspects of Declarative Languages* (PADL 11), 2011, pp. 175–189.
8. G. Salvaneschi et al., "An Empirical Study on Program Comprehension with Reactive Programming," *Proc. 22nd ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (FSE 14), 2014, pp. 564–575.
9. G. Bierman et al., "Pause 'n' Play: Formalizing Asynchronous C#," *ECOOP 2012—Object-Oriented Programming*, LNCS 7313, 2012, pp. 233–257.
10. P. Haller and J. Zaugg, "SIP-22—Async," EPFL, 2013; http://docs.scala-lang.org/sips/pending/async.html.
11. "Asynchronous Programming Patterns," Microsoft; http://msdn.microsoft.com/en-us/library/jj152938.aspx.
12. Y. Lin, S. Okur, and D. Dig, "Study and Refactoring of Android Asynchronous Programming," to be published in *Proc. 2015 Int'l Conf. Automated Software Eng.* (ASE 15), 2015.
13. S. Okur and D. Dig, "How Do Developers Use Parallel Libraries?," *Proc. 20th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, pp. 54–65.
14. S. Cleary, "Best Practices in Asynchronous Programming," Microsoft, 2015; http://msdn.microsoft.com/en-us/magazine/jj991977.aspx.