View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2016 Semih Okur

ENABLING MODERN CONCURRENCY THROUGH PROGRAM TRANSFORMATIONS

BY

SEMIH OKUR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Adjunct Assistant Professor Daniel Dig, Chair
Associate Professor Darko Marinov
Associate Professor Tao Xie
Professor Arie van Deursen, Delft University of Technology

ABSTRACT

Concurrency is becoming the norm in modern software because multicores are now everywhere. Developers use concurrency constructs (i) to make their software responsive and scalable via asynchrony and (ii) to increase the throughput of their software via parallelism. Yet, we know little about how developers use these constructs in practice. Without such knowledge, (i) other developers cannot educate themselves about the state of the practice, (ii) language and library designers are unaware of any misuse, (iii) researchers make wrong assumptions, and (iv) tool providers do not provide the tools that developers really need. We claim that concurrent programming deserves first-class citizenship in empirical research and tool support.

In this dissertation, we study the use, misuse, and underuse of concurrency constructs for C#. Based on our large-scale empirical studies, we found two main problems. First, developers still use complex, slow, low-level, and old constructs. Second, when developers use modern concurrency constructs, they highly misuse them, causing severe consequences such as subtle data-races and swallowed exceptions. Some significantly degrade the performance of async programs or can even deadlock the program. Other misuses make developers erroneously think that their code runs sequentially, when in reality the code runs concurrently; this causes unintended behavior and wrong results.

This dissertation presents four tools to enable C# developers (i) to migrate their software to modern concurrency constructs and (ii) to fix misused modern constructs. First, we present an automated migration tool, TASKIFIER, that transforms old style `Thread` and `ThreadPool` constructs to higher-level `Task` constructs. Second, we present a refactoring tool, SIMPLIFIER, that extracts and converts `Task`-related code snippets into higher-level parallel patterns. Third, we provide ASYNCIFIER to developers who want to refactor their

callback-based asynchronous constructs to new `async/await` keywords. Fourth, we developed AsyncFixer, a static analysis tool that detects and corrects 14 common kinds of async/await misuses that cannot be detected by any existing tools. We released all our tools as plugins for the widely used Visual Studio IDE which has millions of users.

We conducted both quantitative and qualitative evaluation of our tools. We applied the tools thousands of times over the real-world software and we show that they (i) are highly applicable, (ii) are much safer than manual transformations, and (iii) fast enough to be used interactively in Visual Studio. Our tools are useful in practice for both industry and open-source communities. Developers of the open-source software accepted 408 patches which are generated by the tools. We also received positive feedback from the early adopters. In industry, our tools are embraced by two companies. They started to actively use AsyncFixer in their automated build process.

Our empirical findings have (i) educated thousands of developers through our educational web portals and (ii) influenced the design of modern constructs across programming languages. We hope that our dissertation serves as a call to action for the testing, program verification, and program analysis communities to study new constructs from programming languages and standard libraries. Researchers can empirically study these constructs and provide necessary tools to help developers adopt them. Hence, we also present a set of guidelines for researchers that aim to make a practical impact on the usage of new programming constructs.

To my parents, to my lovely wife Hatice Seda and our coming child.

ACKNOWLEDGMENTS

Working on my Ph.D has been a wonderful life experience for me. I deeply appreciate the input and support that I have received from many people through this journey.

Firstly, I would like to express my sincere appreciation to my adviser, Danny Dig, for his endless guidance, support, and encouragement. He introduced me to the world of software refactoring, provided interesting directions for me to explore, and gave me the freedom to investigate my own ideas. He taught me how to be a professional researcher and how to make a positive impact on others. It has been inspirational for me to see how much attention he gives to leadership and growth mindset. Knowing that he cared about me was really precious. Without his continuous support through the most difficult times of my Ph.D study, I would not have been able to accomplish this dissertation.

I would like to thank my committee members, Darko Marinov, Tao Xie, and Arie van Deursen, for their time, kindly help, and insightful comments on my research. They monitored my progress and provided valuable advice.

I am thankful to David Hartveld for the fruitful collaboration resulted in a distinguished paper award; Yu Lin for the collaboration on conducting a formative study for Android. Special thanks to my dear friend and officemate Cosmin Radoi for helping me explore research ideas and prepare for my qualification and preliminary exams. In addition to technical discussions, we have had wonderful chats about the academic life.

Without industry partners, my Ph.D study would not have created such an impact. My two internships at Microsoft taught me to value the things that are really important for developers. Thanks to Wolfram Schulte, Nikolai Tillmann, Stephen Toub, Dustin Campbell, Don Syme, Jon Skeet, and Neal Gafter for providing new perspectives on my research.

I have learned a lot from discussions with my colleagues, their comments on various drafts

of papers that make up this dissertation, and their feedback during practice talks. I am grateful to Mihai Codoban, Michael Hilton, Caius Brindescu, Alex Gyori, Stas Negara, Mohsen Vakilian, Milos Gligoric, Samira Tasharofi, Sergey Shmarkatyuk, and anonymous reviewers. Also, I would like to thank all my dear friends and colleagues at the University of Illinois.

My parents deserve the most special of thanks for their endless love and having faith in me for all these years. I would like to extend my most sincere gratitude and appreciation to my father, Ibrahim, my mother, Gonul, and my brother, Yasin Okur.

Hatice Seda, my dear and lovely wife, has been my greatest supporter and my best friend in life. It is her tremendous love that encourages me to be positive and brave, and supports me to finish this dissertation. I do not have enough words to express my gratitude for all that she means to me.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER 1 Introduction 1
- 1.1 Problem Description 1
- 1.2 Thesis Statement 2
- 1.3 Migration to Higher-Level Concurrency Constructs 3
- 1.4 Refactoring to `async/await` Keywords 5
- 1.5 Finding and Correcting Misused Asynchrony 7
- 1.6 Organization 9

CHAPTER 2 Background on Concurrent Programming 10
- 2.1 Concurrency, Parallelism, and Asynchrony 10
- 2.2 Concurrent Programming in C# 11
- 2.3 Definitions 23

CHAPTER 3 Migration to Higher-Level Concurrency Constructs 24
- 3.1 Introduction 24
- 3.2 Formative Study 27
- 3.3 TASKIFIER: Migrating `Thread`-based Code to `Task` 30
- 3.4 SIMPLIFIER: Refactoring to Parallel Design Patterns 37
- 3.5 Evaluation 46
- 3.6 Summary 50

CHAPTER 4 Refactoring to `async/await` Keywords 51
- 4.1 Introduction 51
- 4.2 Formative Study 53
- 4.3 How do Developers Use Asynchronous Programming? 54
- 4.4 Do Developers Misuse `async/await`? 59
- 4.5 ASYNCIFIER: Refactoring from Callbacks to `async/await` 63
- 4.6 ASYNCFIXER: Fixing Common Misuses 73
- 4.7 Evaluation 75
- 4.8 Discussion 78
- 4.9 Summary 82

CHAPTER 5 Finding and Correcting Misused Asynchrony 83
5.1 Introduction 83
5.2 Formative Study 85
5.3 AsyncFixer: Extending it to Fix 10 New Misuses 88
5.4 Performance Misuses 89
5.5 Correctness Misuses 94
5.6 Bad Practices 100
5.7 Evaluation 101
5.8 Discussion 104
5.9 Summary 106

CHAPTER 6 A Guide for Researchers Studying New Constructs 107
6.1 Choosing a Construct to Study 107
6.2 Analyzing the Usage from Early Adopters 109
6.3 Implementing the Tools that Developers Need 110
6.4 Evaluating the Tools 112
6.5 Interactions for Impact 113

CHAPTER 7 Related Work 116
7.1 Empirical Studies for Language and API Usage 116
7.2 Empirical Studies for Concurrency Bugs 117
7.3 Concurrency Misuse Detection 118
7.4 Refactoring for Concurrency 119

CHAPTER 8 Conclusion and Future Work 121
8.1 Revisiting Thesis Statement 121
8.2 Understanding Runtime Properties of Concurrent Code 123
8.3 Runtime Implications of `async/await` 124
8.4 Context Decider Tool 125
8.5 Guiding Programmers to Parallelize the Code 126

REFERENCES 128

LIST OF TABLES

3.1 Usage of parallel idioms 29
3.2 Applicability of Taskifier and Simplifier 47

4.1 Usage of asynchronous idioms in the proprietary code 56
4.2 Usage of asynchronous idioms in the open-source code 56
4.3 Most popular I/O operations used in the open-source code 57
4.4 Catalog of I/O operations used in the open-source code 57
4.5 Statistics of `async/await` misuses in the open-source code 63
4.6 AsyncFixer patches accepted by developers 78

5.1 Usage trend of `async/await` over four years 86
5.2 Misuse statistics of `async/await` 103

LIST OF FIGURES

2.1 Where is callback-based APM code executing? 19
2.2 Where is the async/await-based code executing? 21

4.1 AsyncFixer tool interactivity 73

5.1 Usage trend of `async` I/O constructs over four years 86
5.2 Usage trend of `async` CPU-bound constructs over four years 87

6.1 Guide for researchers studying new programming constructs 108

CHAPTER 1

Introduction

1.1 Problem Description

The computing hardware industry has resorted to multicore CPUs in order to keep up with the previous prediction of Moore's law. While the number of transistors will keep doubling, the multicore revolution puts pressure on software developers to use concurrency if they want to benefit from future hardware improvements. Because multicores are now everywhere, concurrency is becoming the norm in modern software. Developers use concurrency (i) to make their software responsive and scalable via asynchrony and (ii) to increase the throughput of their software via parallelism. However, developers find it challenging to introduce asynchrony and parallelism to their software.

The industry leaders hope to simplify concurrent programming by providing concurrency libraries (APIs). Microsoft provides Task Parallel Library (TPL) [1], Parallel Language Integrated Query (PLINQ) [2], and Threading [3] for .NET languages (e.g., C#). Java developers uses `java.util.concurrent` package. Intel provides Threading Building Blocks (TBB) [4] for C++. Despite syntactic differences, these libraries provide similar features such as scalable concurrent collections, high-level parallel constructs, and lightweight tasks.

Yet, we know little about how developers use these libraries in practice. Empirical studies that answer usage questions are desperately needed. Without such knowledge, other developers cannot educate themselves about the state of the practice, language and library designers are unaware of any misuse, researchers make wrong assumptions, and tool providers do not provide the tools that developers really need. We believe that concurrent programming deserves first-class citizenship in empirical research and tool support.

Based on our empirical studies regarding the usage of concurrency, we found two main

problems. First, developers still use complex, slow, low-level, and old constructs. For instance, `Thread` is still the primary choice for most developers. They heavily use callback-based asynchronous constructs, not benefitting from new language keywords, `async/await`. Second, when developers use modern concurrency constructs, they highly misuse them. These misuses have severe consequences. Some significantly degrade the performance of async programs or can even deadlock the program. Other misuses make developers erroneously think that their code runs sequentially, when in reality the code runs concurrently; this causes unintended behaviour and wrong results.

This dissertation presents four tools to enable developers to correctly use modern concurrency in C# code. First, we present an automated migration tool, TASKIFIER, that transforms old style `Thread` and `ThreadPool` constructs to higher-level `Task` constructs. Second, we present a refactoring tool, SIMPLIFIER, that extracts and converts `Task`-related code snippets into higher-level parallel patterns. Third, we provide ASYNCIFIER to developers who want to refactor their callback-based asynchronous constructs to new `async/await` keywords. Fourth, we developed ASYNCFIXER, a static analysis tool that detects and corrects 14 common kinds of `async/await` misuses.

1.2 Thesis Statement

Our thesis is three-pronged:

(1) Empirical studies about concurrency constructs can provide insights into how developers use, misuse, or underuse concurrency.
(2) It is possible to design and develop interactive program transformations that enable developers (i) to migrate their software to modern concurrency constructs and (ii) to fix misused modern constructs.
(3) These program transformations are useful in practice for both industry and open-source communities.

To prove this thesis statement, we conducted three main studies: (1) an empirical study on the usage of old and modern C# concurrency constructs and program transformations for

safely migrating legacy software to modern constructs, (2) an empirical study to understand the usage of asynchronous programming and a refactoring technique for converting callback based asynchronous constructs to `async/await` keywords, (3) an empirical study on the misuses of `async/await` keywords and program transformations to fix them. For each program transformation, we submitted patches to developers (a total of 408 patches) and they think our patches are useful.

The rest of this chapter introduces these three bodies of research.

1.3 Migration to Higher-Level Concurrency Constructs

In the quest to support programmers with faster, more scalable, and readable code, parallel libraries continuously evolve from low-level to higher-level abstractions. In the C# ecosystem, .NET 1.0 (2002) supported a threading library, .NET 4.0 (2010) added lightweight tasks, declarative parallel queries, and concurrent collections, .NET 4.5 (2012) added reactive asynchronous operations.

Low-level abstractions, such as `Thread`, make parallel code complex, unscalable, and slow. Because `Thread` represents an actual OS-level thread, it consumes a non-trivial amount of memory, and starting and cleaning up after a retired thread takes hundreds of thousands of CPU cycles. When developers mix old and new parallel abstractions in their code, it makes it hard to reason about the code because all these abstractions have different scheduling rules. On the other hand, higher-level abstractions such as .NET `Task` [5], a unit of parallel work, make the code less complex. `Task` gives advanced control to the developer (e.g., chaining, cancellation, futures, callbacks), and is more scalable than `Thread`. Unlike threads, tasks are *lightweight*: they have much smaller performance overhead and the runtime system automatically balances the workload.

However, most developers are oblivious to the benefits brought by the higher-level parallel abstractions. We and other researchers [6, 7] found that `Thread` is still the primary choice for most developers. Therefore, a lot of code needs to be migrated from low-level parallel abstractions to their higher-level equivalents. However, this migration is (i) tedious and (ii) error-prone, e.g., it can harm performance and silence the uncaught exceptions. We present

an automated migration tool, TASKIFIER, that transforms old style `Thread` and `ThreadPool` abstractions to higher-level `Task` abstractions in C# code.

The recent versions of concurrency libraries provide even higher-level abstractions on top of Tasks. For example, the `Parallel` abstraction in C# supports parallel programming design patterns: data parallelism in the form of parallel loops, and fork-join task parallelism [8] in the form of parallel tasks co-invoked in parallel. These dramatically improve the readability of the parallel code. However, developers rarely use them and developers are not aware of them. We present another tool, SIMPLIFIER, that extracts and converts Task-related code snippets into higher-level parallel patterns.

This research makes the following contributions:

1. Problem: To the best of our knowledge, this is the first study that describes the novel problem of migrating low-level parallel abstractions into their high-level counterparts. We show that this problem appears in real-life applications by bringing evidence of its existence from a corpus of 880 C# open-source applications.

2. Algorithms: We describe the analysis and transformation algorithms which address the challenges of (i) migrating `Thread`-based usage into `Task` abstractions and (ii) transforming `Task` code snippets into higher-level parallel design patterns.

3. Tools: We implemented our algorithms into two tools, TASKIFIER and SIMPLIFIER. We implemented them as extensions to Visual Studio, the primary IDE for C#.

4. Evaluation: We empirically evaluated our implementations by using our code corpus of 880 C# applications. We applied TASKIFIER 3026 times and SIMPLIFIER 405 times. First, the results show that the tools are widely *applicable*. Second, these transformations are *valuable*: TASKIFIER reduces the size of the converted code snippets by 2617 SLOC and SIMPLIFIER reduces by 2420 SLOC in total. Third, the tools save the programmer from manually changing 10991 SLOC for the migration to `Task` and 7510 SLOC for the migration to `Parallel`. Fourth, automated transformations are safer. Several of the manually written `Task`-based codes by open-source developers contain problems: 32% are using blocking operations in the body of the `Task`, which can result in thread-starvation. Fifth, open-source developers found our transformations useful. We submitted 66 patches generated by our tools and the open-source developers accepted 53.

1.4 Refactoring to `async/await` Keywords

Concurrency has two different forms: *parallelism* and *asynchrony*. When developers use concurrency for parallelism, it is to improve a non-functional requirement such as performance and throughput. When developers use concurrency for asynchrony, it is to make a system more scalable and responsive.

Nowadays, asynchronous programming is in demand because responsiveness is increasingly important on all modern devices: desktop, mobile, or web apps. Therefore, major programming languages have APIs that support non-blocking, asynchronous operations (e.g., to access the web, or for file operations). While these APIs make asynchronous programming possible, they do not make it easy because they rely on callbacks. Callbacks invert the control flow, are awkward, and obfuscate the intent of the original synchronous code [9].

Recently, major languages (F# [9], C# and Visual Basic [10], and Scala [11]) introduced `async/await` language keywords that resemble the straightforward coding style of traditional synchronous code. Thus, they recognize asynchronous programming as a first-class citizen. Yet, we know little about how developers use asynchronous programming and specifically the new `async` constructs in practice.

We present the first study that analyzes the usage of asynchronous libraries and new language constructs, `async` in both industry and open-source communities. We analyze 1378 open-source Windows Phone (WP) apps, comprising 12M SLOC (C#), produced by 3376 developers. In industry, we analyze 3.2M SLOC, comprising various platforms written in C# (console, web, and library) at two companies.

We found that developers heavily use callback-based asynchronous idioms. However, Microsoft officially no longer recommends these asynchronous idioms [12] and has started to replace them with new idioms in new libraries (e.g., WinRT). Developers need to refactor callback-based idioms to new idioms that can take advantage of the `async` keywords. The changes that the refactoring requires are non-trivial, though. For instance, developers have to inspect deep call graphs. Furthermore, they need to be extra careful to preserve exception handling behavior. Thus, we implemented the refactoring as an automated tool, Asyncifier.

We also found that nearly half of open-source WP8 apps have started to use the 9-month-old `async` keywords in 2013. However, developers frequently misuse `async` in both industry and open-source codebases. We define *misuse* as anti-patterns [13], which hurt performance and might cause serious problems like deadlocks. For instance, we found that 14% of methods that use (the expensive) `async` keywords do this unnecessarily, 19% of methods do not follow an important good practice [14], 1 out of 5 apps misses opportunities in async methods to increase asynchronicity, and developers (almost) always unnecessarily capture context, hurting performance. Thus, we implemented a transformation tool, AsyncFixer, that finds and corrects the misused `async`.

This study makes the following contributions:

1. Formative Study: To the best of our knowledge, this is the first large-scale empirical study to answer questions about asynchronous programming and `async`, that will be available soon in other major programming languages. We present implications of our findings from the perspective of four main audiences: developers, language and library designers, researchers, and tool providers.

2. Toolkit: We implemented the analysis and transformation algorithms to address the challenges: Asyncifier and AsyncFixer for Visual Studio.

3. Evaluation: We evaluated our tools by using the code corpus and applied the tools hundreds of times. 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. Using AsyncFixer, we found and reported misuses in 19 apps. 19 replied and accepted each of our 286 patches.

4. Outreach: Because developers learn new language constructs through both positive and negative examples, we designed an education portal, `http://LearnAsync.NET`, where developers can find hundreds of real-world examples of all asynchronous idioms and inspect real-world misuse examples of `async` keywords.

1.5 Finding and Correcting Misused Asynchrony

In 2013, we were the first to study [15] the novel `async/await` in C#. Back then, the main problem was *under-use*: programs used it sparingly, and it was not popular outside the domain of mobile apps. This inspired us to develop a refactoring tool [15] that helps developers embrace it. In our latest formative study comprising 51M SLOC representing all domains of computing, we found a 10-fold increase in usage of `async/await` since 2013. By 2016, developers across all domains embraced `async/await`. Are developers now using `async/await` correctly?

In this research, we first conduct a large scale formative study to understand how developers use `async/await` in both open-source and proprietary programs across all domains. Our formative study answers two research questions:

RQ1: *How are developers embracing modern asynchronous constructs?* We analyzed annual code snapshots of the 100 most popular C# repositories in Github [16] from 2013 to 2016, comprising 51M SLOC in total. We found that developers fully embraced modern, `async/await`-based constructs, surpassing the usage of legacy, callback-based constructs.

RQ2: *To what extent do developers misuse modern asynchronous constructs?* We discovered 10 kinds of prevalent misuses that have not been studied before.

These `async/await` misuses have severe consequences. Some cause subtle data-races that are unique to `async/await` and cannot be detected by previous race detecting techniques. Other misuses "swallow" the run-time uncaught exceptions thrown by a program which hinder developers when searching for errors. Other misuses significantly degrade the performance of async programs or can even deadlock the program. Other misuses make developers erroneously think that their code runs sequentially, when in reality the code runs concurrently; this causes unintended behaviour and wrong results.

The `async/await` misuses are a new source of bugs in software that the research community has not studied before. Moreover, most of the misuses that we discovered cannot be detected by the existing bug-finding tools. This study is the first to raise the awareness of the research

community on an increasingly important source of bugs. We hope that it serves as a call to action for the testing, program verification, and program analysis community.

Inspired by our formative study, we extended AsyncFixer, our existing static analysis tool, (i) *to detect* 10 more kinds of `async/await` misuses, and (ii) *to fix* them via program transformations. Developers can use AsyncFixer as *live* code analysis: it constantly scans the source code as soon as the user types in new code. A key challenge is to make the detection efficient so that developers can use AsyncFixer interactively. The static analysis that detects the 10 kinds of misuses is non-trivial: it requires control- and data-flow, as well as dependence information. A syntactic find-and-replace tool cannot detect these misuses.

To empirically evaluate AsyncFixer, we ran it over a diverse corpus of 500 programs from GitHub, comprising 24M SLOC. We also ran it over the code-bases of two industrial partners, comprising 4M SLOC. AsyncFixer detected 891 previously unknown misuses in open-source software, and 169 in proprietary software. The open-source developers accepted 41 AsyncFixer-generated patches. Moreover, our industrial partners integrated AsyncFixer directly into their automated build process, so that misuses are caught and fixed before they are in production code.

This study makes the following contributions:

1. Formative study: To the best of our knowledge, we are the first to study the usage and evolution of modern asynchronous constructs such as `async/await` over a diverse corpus. We discovered 10 kinds of prevalent misuses of `async/await`.

2. Algorithms: We designed the analysis and transformation algorithms to detect and fix 10 kinds of `async/await` misuses.

3. Tool: We implemented the algorithms in our existing static analysis tool, AsyncFixer which integrates with Visual Studio.

4. Evaluation: By running AsyncFixer over a hybrid open-source and proprietary corpus of 28M SLOC, we show that it is applicable and efficient. Open-source developers accepted 41 patches and our industrial partners are now using AsyncFixer in their automated build process. This shows AsyncFixer is useful.

5. Outreach: To increase our practical impact, we improved our existing educational portal, `http://LearnAsync.NET`, which shows real-world examples of `async/await` misuses.

1.6 Organization

.

The rest of the dissertation is organized as follows: Chapter 2 gives background information on concurrent programming. First, it explains the concepts of concurrency, asynchrony, and parallelism. Second, it gives a detailed overview of concurrency constructs (both libraries and language keywords) in C#.

Chapter 3 presents the design and evaluation of two automated tools: (i) TASKIFIER converts `Thread`-based usage to lightweight `Task`, (ii) SIMPLIFIER converts `Task`-based code into higher-level parallel design patterns. Our empirical evaluation shows that the tools are highly applicable, reduce the code bloat, and are much safer than manual transformations.

Chapter 4 presents the design and evaluation of two automated tools: (i) ASYNCIFIER refactors callback-based asynchronous code to use `async`, (ii) ASYNCFIXER finds and corrects common misuses of `async`. Evaluation shows that these tools are applicable, efficient, and useful. Developers accepted 408 patches generated by our tools.

Chapter 5 presents 10 kinds of prevalent misuses of `async/await`, which have severe consequences such as subtle data-races, swallowed exceptions, and unintended behavior. We extend the capability of our previous tool, ASYNCFIXER so that it detects and corrects these misuses as well. ASYNCFIXER discovered 891 misuses in open-source and 169 misuses in proprietary code-bases. Our industrial partners adopted our tool in their automated build process.

Chapter 6 presents a set of guidelines for researchers that aim to make a practical impact on the usage of new programming constructs.

Chapter 7 puts the dissertation in the context of the related research. Chapter 8 concludes and talks about future work.

Parts of this dissertation have been published in technical reports and conferences. In particular, Chapter 3 is described in our ECOOP-2014 [19] paper and Chapter 4 in our ICSE-2014 paper [15], which received *ACM SIGSOFT Distinguished Paper Award*. Chapter 5 is currently under review at a premiere software engineering conference. These chapters have been extended and revised when writing this dissertation.

CHAPTER 2

Background on Concurrent Programming

2.1 Concurrency, Parallelism, and Asynchrony

The computing hardware industry has resorted to multi-core CPUs in order to keep up with the previous prediction of Moore's law. While the number of transistors will keep doubling, the multi-core revolution puts pressure on software developers to use concurrency if they want to benefit from future hardware improvements. Developers use concurrency to improve the performance and scalability of software through enhancing its throughput and responsiveness on multiprocessor platforms.

Concurrency, parallelism, and asynchrony are three related concepts, but are slightly different. Concurrency is a conceptual property of a system representing the fact that multiple activities may run during overlapping time periods. However, parallelism and asynchrony are runtime states of a concurrent system, hence they are forms of concurrency.

When developers use concurrency for *parallelism*, it is to improve a non-functional requirement such as performance and throughput. They use multiple threads that run on multicore machine so that multiple operations run at the same time, in parallel. Parallelism works well when you can separate a task into independent pieces of work. For example, the manipulation of a large set of data can sometimes be divided over multiple cores, often in equal operations. Parallelism literally physically run multiple operations, at the same time using multi-core infrastructure of CPU, by assigning one core to each operation. In a single-core CPU, parallelism cannot be achieved.

When developers use concurrency for *asynchrony*, it is to make a system more scalable and responsive. The emphasis lies on computations that run in the background from the perspective of a specific thread of execution. For example, to keep the UI responsive, devel-

opers asynchronously execute expensive and time-consuming operations so that the event dispatch thread becomes available to respond to UI events (e.g., scrolling, clicking on a button). Asynchrony is possible in a single-core CPU due to context-switching between threads.

This dissertation uses the term of *concurrency* as an umbrella term that encompasses both *asynchrony* and *parallelism*.

2.2 Concurrent Programming in C#

Developers need to use concurrency to leverage multi-core technology. However, it is known that concurrent programming is hard [20]. The industry leaders hope to convert the hard problem of using concurrency into the easier problem of using concurrency libraries. Microsoft provides several libraries for .NET languages (e.g., C#). Java developers uses `java.util.concurrent` [21] package. Intel provides Threading Building Blocks (TBB) [4] for C++. Despite syntactic differences, these libraries provide similar features such as threads, scalable concurrent collections (e.g., `ConcurrentDictionary`), high-level parallel constructs (e.g., `Parallel.For`), and lightweight tasks (e.g., futures, promises). Their runtime systems also provide automatic load balancing [8].

C# is particularly rich when it comes to concurrency libraries. Microsoft provides concurrency constructs with four different libraries in .NET framework: Task Parallel Library (TPL) [1], Parallel Language Integrated Query (PLINQ) [2], Concurrent Collections [22], and Threading [3]. We categorized concurrency constructs into two main parts: (i) constructs for offloading CPU-bound operations to another thread, (ii) constructs for offloading I/O-bound operations to another thread.

2.2.1 Concurrency constructs for CPU-bound operations

C# provides five main constructs to offload CPU-bound or custom operations to another thread so that the current thread is not blocked: (1) starting a new `Thread`, (2) queuing a job in the `ThreadPool`, (3) starting execution of a `BackgroundWorker`, (4) running a `Task`,

(5) calling methods of `Parallel` class: `Parallel.For`, `Parallel.Invoke`.

`Thread` is the primary construct for encapsulating concurrent computation, and `ThreadPool` allows one to reuse threads. `Task` and `Parallel` constructs are considered modern constructs, whereas the rest represents the legacy constructs. Microsoft now encourages developers to use `Task` in order to write scalable, hardware independent, fast, and readable asynchronous code [23].

Thread class. Operating systems use processes to separate the different applications that they are executing. *Thread* is the basic unit to which an operating system allocates processor time, and more than one thread can be executing code inside one process. Threading library in .NET provides an abstraction of threads, `Thread` class since its first version, 2003.

`Thread` represents an actual OS-level thread, so it is expensive to use; creating a `Thread` needs about 1.5 MB memory space. Windows also creates many additional data structures to work with this thread, such as a Thread Environment Block (TEB), a user mode stack, and a kernel mode stack. Bringing in a new thread may also mean more thread context switching, which further hurts performance. It takes about 200,000 CPU cycles to create a new thread, and about 100,000 cycles to retire a thread.

On one hand, `Thread` class allows the highest degree of control; developers can set many thread-level properties like the stack size, priority, background and foreground. However, general-purpose apps do not need most of these low-level features. On that matter, Microsoft discourages developers to use these features because they are usually misused [23]. In modern C# code, developers should rarely need to explicitly start their own thread.

On the other hand, `Thread` has some limitations. For example, a `Thread` constructor can take at most one parameter and this parameter must be of type `Object`. In Code 2.1, a `Thread` is first created with its body which is `MailSlotChecker` method. `ParameterizedThreadStart` indicates that this method needs to take a parameter. After priority and background properties are set, the parameter, `info` is created and given to `Start` method that asynchronously executes the `Thread`. When the instance `info` of `MailSlotThreadInfo` type is passed to `Thread` body, it will be forced to upcast to `Object` type. Developers manually need to downcast it to `MailSlotThreadInfo` type

in `MailSlotChecker` method. Hence, this introduced verbose code like explicit casting, `ParameterizedThreadStart` objects. To wait for the termination of the `Thread`, the code invokes a blocking method, `Join`.

Code 2.1 Thread usage example from Tiraggo[24] app

```
Thread thread = new Thread(new ParameterizedThreadStart(MailSlotChecker));
thread.Priority = ThreadPriority.Lowest;
thread.IsBackground = true;
MailSlotThreadInfo info = new MailSlotThreadInfo(channelName, thread);
thread.Start(info);
...
thread.Join(info);
```

ThreadPool class. There is no need to create or destroy threads for each work item; the threads are recycled in the pool. .NET provides an abstraction, the `ThreadPool` class, since its first version.

Although `ThreadPool` class is efficient to encapsulate concurrent computation, it gives developers no control at all. Developers only submit work which will execute at some point. The only thing they can control about the pool is its size. `ThreadPool` offers no way to find out when a work item has been completed (unlike `Thread.Join()`), neither a way to get the result.

Code 2.2 shows two main examples of `ThreadPool` usage. `QueueUserWorkItem` is used to put work items to the thread pool. The first example executes `foo(param)` method call in the thread pool but it is unclear because of the syntax. The second example executes the same thing with a lambda function which is introduced in C# 4.0. Developers can directly pass the parameters to the lambda function. However, `QueueUserWorkItem` only accepts a lambda function that takes one parameter: (e.g., `(x)=> foo(x)`. Developers always need to provide one parameter, regardless of whether they use it or not, thus many times they call this parameter `unused` or `ignored`.

Code 2.2 ThreadPool example

```
ThreadPool.QueueUserWorkItem(new WaitCallback(foo),param);
ThreadPool.QueueUserWorkItem((unused)=> foo(param));
```

Task class. `Task` was introduced in the Task Parallel Library [25] with the release of .NET 4.0 in 2010. `Task` offers the best of both worlds: `Thread` and `ThreadPool`. `Task` is simply a lightweight thread-like entity that encapsulates an asynchronous operation. Like `ThreadPool`, a `Task` does not create its own OS thread so it does not have high-overhead of `Thread`. Instead, it is executed by a `TaskScheduler`; the default scheduler simply runs on the thread pool. `TaskScheduler` use work-stealing techniques which are inspired by the Java fork-join framework [8].

`Task` is a lightweight thread-like entity that encapsulates an asynchronous operation. Using tasks instead of threads has many benefits [25] - not only are tasks more efficient, they also abstract away from the underlying hardware and the OS specific thread scheduler. `Task<T>` is a generic class where the associated action returns a result; it essentially encapsulates the concept of a "Future" computation. `TaskFactory` creates and schedules tasks. Here is a fork/join task example from the *passwordgenerator* [26] application:

Code 2.3 Task example

```
for (uint i = 0; i < tasks.Length; i++)
{
  tasks[i] = tf.StartNew(() => GeneratePassword(length, forceNumbers, ...), _cancellation.Token);
}
Task.WaitAll(tasks, _cancellation.Token);
```

The code creates and spawns several tasks stored in an array of tasks (the fork step), and then waits for all tasks to complete (the join step).

Unlike the `ThreadPool`, `Task` also allows developers to find out when it finishes, and (via the generic `Task<T>`) to return a result. A developer can call `ContinueWith()` on an existing `Task` to make it run more code once the task finishes; if it is already finished, it will run the callback immediately. A developer can also synchronously wait for a task to finish by calling `Wait()` (or, for a generic task, by getting the Result property). Like `Thread.Join()`, this will block the calling thread until the task finishes.

The bottom line is that `Task` is almost always the best option; it provides a much more powerful API and avoids wasting OS threads. All newer high-level concurrency APIs, including PLINQ, `async` language features, and modern asynchronous methods are all built

on `Task`. It is becoming the foundation for all parallelism, concurrency, and asynchrony in .NET. According to Microsoft, `Task` is the only preferred way to write multithreaded and parallel code [23].

Parallel class. `Parallel` class is a part of the TPL library as well. It provides three main methods to support parallel programming design patterns: data parallelism (via `Parallel.For` and `Parallel.ForEach`), and task parallelism (via `Parallel.Invoke`).

`Parallel.For` method accepts three parameters: an inclusive lower-bound, an exclusive upper-bound, and a lambda function to be invoked for each iteration. By default, it uses the work queued to .NET thread pool to execute the loop with as much parallelism as it can muster. For example, `Parallel.For(0, n, (i)=> foo(i))`.

`Parallel.ForEach` is a very specialized loop. Its purpose is to iterate through a specific kind of data set, a data set made up of numbers that represent a range. For example, `Parallel.ForEach(books, (book)=>foo(book))`.

`Parallel.Invoke` runs the operations (lambda functions) given as parameters concurrently and waits until they are done. It parallelizes the operations, not the data. For example, `Parallel.Invoke(()=> foo(), ()=> boo())`.

`Parallel` class works efficiently even if developers pass in an array of one million lambda functions to `Parallel.Invoke` or one million iterations to `Parallel.For`. This is because `Parallel` class does not necessarily use one `Task` per iteration or operation, as that could add significantly more overhead than is necessary. Instead, it partitions the large number of input elements into batches and then it assigns each batch to a handful of underlying tasks. Under the covers, it tries to use the minimum number of tasks necessary to complete the loop (for `For` and `ForEach`) or operations (for `Invoke`) as fast as possible. Hence, Microsoft shows that `Parallel` class performs faster than equivalent `Task`-based code in some cases [27].

`Parallel` class will run iterations or operations in parallel unless this is more expensive than running them sequentially. The runtime system handles all thread scheduling details, including scaling automatically to the number of cores on the host computer.

Here is a usage example from the *ravendb* [28] application:

Code 2.4 Parallel.For example

```
Parallel.For(0, 10, counter => { ... ProcessTask(counter, database, table)} )
```

The first two arguments specify the iteration domain, and the third argument is a C# lambda function called for each iteration. TPL also provides more advanced variations of `Parallel.For`, useful in map/reduce computations.

PLINQ library. .NET 4.0 provides a fourth parallel library, the Parallel Language-Integrated Query (PLINQ) library, which supports a declarative programming style. PLINQ queries operate on `IEnumarable` objects by calling `AsParallel()`. Here is an example from the *AppVisum* [29] application:

Code 2.5 PLINQ example

```
assembly.GetTypes().AsParallel()
  .Where(t => t.IsSubclassOf(typeof(ControllerBase)))
  .Select(t => new ...)
  .ForAll(t => controllersCache.Add(t.Name, t.Type));
```

After the `AsParallel`, the data is partitioned to worker threads, and each worker thread executes in parallel the following `Where`, `Select`, and `ForAll`.

2.2.2 Concurrency constructs for IO-bound operations

I/O operations are more complicated to offload asynchronously. The naive approach would be to just start another thread to run the synchronous operation asynchronously, using the same mechanics as used for CPU-bound code. This approach would help the event dispatch thread become available, but it hurts scalability. The spawned thread will be busy-waiting for the synchronous I/O operation. Because a new thread is a valuable resource, consuming it just for busy-waiting hurts scalability.

The solution is to use asynchronous APIs provided by the .NET framework, which take advantage of operating system's support for *I/O completion ports*. I/O completion ports provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system. When a process creates an I/O completion port, the system creates

an associated queue object for requests whose sole purpose is to service these requests. Under the hood, that queue object sets up the plumbing to get the port to automatically notify the process when the I/O operation completes. The core idea is that no thread is waiting while the I/O operation takes place.

The .NET framework mainly provides three models for asynchronous programming, which use I/O completion threads: (1) callback-based, Asynchronous Programming Model (APM) in 2002, (2) callback-based, Event-based Asynchronous Pattern (EAP) in 2005, (3) task-based, Task Asynchronous Pattern (TAP) in 2012.

Both APM and EAP represent the legacy models, whereas TAP is now considered the modern model for async I/O operations. For (almost) each I/O operation, .NET provides constructs for each model. For instance, to download the requested resource as a string, .NET provides `BeginDownloadString` for APM, `DownloadStringAsync` for EAP, and `DownloadStringTaskAsync` for TAP.

UI freezes without concurrency. Modern graphical user interface (GUI) frameworks use an event dispatch thread that is a single, specific thread to manipulate the user interface (UI) state [30, 31, 32, 33]. The UI freezes when this single thread is busy with long-running CPU-bound or blocking I/O operations. Code 2.6 shows an example of a frozen event handler: `Button_Click`. When a button click 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. Code 2.6 shows an example of such an event handler, method `Button_Click`. It uses the `GetFromUrl` method to download the contents of a URL, and place it in a text box. Because `GetFromUrl` is waiting for the network operation to complete, the UI event thread is blocked, and the UI is unresponsive.

Keeping UIs responsive thus means keeping the UI event thread free of those long-running or blocking operations. If these operations are executed asynchronously in the background, the foreground UI event thread does not have to busy-wait for completion of the operations. That frees up the UI event thread to respond to user input, or redraw the UI: the user will experience the UI to be responsive.

Asynchronous programming model (APM). APM, the Asynchronous Programming

Code 2.6 Synchronous example

```
1  void Button_Click(...) {
2    string contents = GetFromUrl(url)
3    textBox.Text = contents;
4  }
5  string GetFromUrl(string url) {
6    WebRequest request = WebRequest.Create(url);
7    WebResponse response = request.GetResponse();
8    Stream stream = response.GetResponseStream();
9    return stream.ReadAsString();
10 }
```

Code 2.7 APM-based example

```
1  void Button_Click(...) {
2    GetFromUrl(url);
3  }
4  void GetFromUrl(string url) {
5    var request = WebRequest.Create(url);
6    request.BeginGetResponse(Callback, request);
7  }
8  void Callback(IAsyncResult aResult) {
9    var request = (WebRequest)aResult.AsyncState;
10   var response = request.EndGetResponse(aResult);
11   var stream = response.getResponseStream();
12   var content = stream.ReadAsString();
13   Dispatcher.BeginInvoke(() => {
14       textBox.Text = content;
15   });
16 }
```

Model, was part of the first version of the .NET framework, and 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 Code 2.7, `BeginGetResponse` is such a `Begin` method, and `EndGetResponse` is an `End` method.

`BeginGetRepsonse` is used to initiate 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, which can then continue to do something else. 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. See Figure 2.1 for an illustration of this flow of events.

The APM `Begin` method has two pattern-related parameters. The first parameter is the callback delegate (which is a managed, type-safe equivalent of a function pointer). It can be

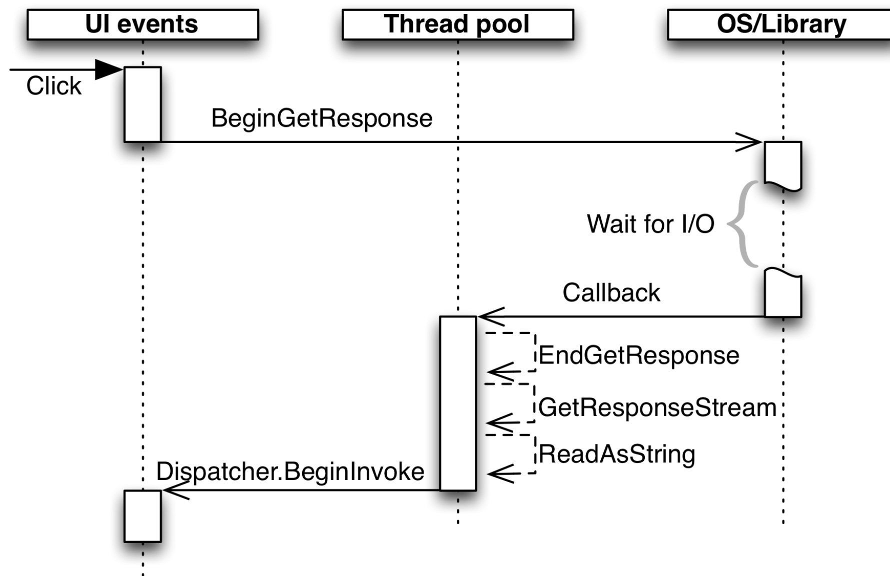

Figure 2.1: Where is callback-based APM code executing?

defined as either a method reference, or a lambda expression. The second parameter allows the developer to pass any single object reference to the callback, and is called `state`.

The .NET framework will execute the callback delegate on the thread pool once the asynchronous background operation completes. The `EndGetResponse` method is then used in the callback to obtain the result of the operation, the actual `WebResponse`.

Note a subtle difference between the synchronous, sequential example in Code 2.6 and the asynchronous, APM-based example in Code 2.7. 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.

Task-based Asynchronous Pattern (TAP). TAP, the Task-based Asynchronous Pattern, provides for a slightly different approach. TAP methods have the same base operation name as APM methods, without 'Begin' or 'End' prefixes, and instead have an 'Async' suffix. The API consists of methods that start the background operation and return a `Task` object. The `Task` represents the operation in progress, and its future result.

The `Task` can be (1) queried for the status of the operation, (2) synchronized upon to wait for the result of the operation, or (3) set up with a continuation that resumes in the background when the task completes (similar to the callbacks in the APM model).

2.2.3 Pause and play with `async/await`

Drawbacks of APM and plain TAP. Using APM and plain TAP directly 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. For APM, 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 Code 2.7, lines 9-10. 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 Code 2.7, line 13-15.

`async/await` keywords. C# 5 introduced `async/await` keywords in 2012 to treat asynchronous programming as a first-class citizen. These new keywords can only be used with modern constructs: methods for running a custom `Task` (e.g., `Task.Run()`) and TAP methods for async I/O operations (e.g., `GetResponseAsync()`). The return type of both types of methods is a `Task`, which is a requirement of using the new keywords.

`Task` represents the operation in progress, and its future result. It can be (1) queried for the status of the operation, (2) synchronized upon to wait for the result of the operation, or (3) set up with a continuation that resumes in the background when the task completes.

To solve this problem, the `async` and `await` keywords have been introduced in 2012 in C# 5.0. 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 `Task`'s background operation is completed, the method is resumed from right after the `await` expression. Code 2.8 shows the TAP- & `async`-based equivalent of Code 2.7, and Figure 2.2

Code 2.8 TAP & `async/await`-based example

```
1  async void Button_Click(...) {
2    var content = await GetFromUrlAsync(url);
3    textBox.Text = content;
4  }
5  async Task<string> GetFromUrlAsync(string url) {
6    var request = WebRequest.Create(url);
7    var response = await request.GetResponseAsync()
8                     .ConfigureAwait(false);
9    var stream = response.GetResponseStream();
10   return stream.ReadAsString();
11 }
```

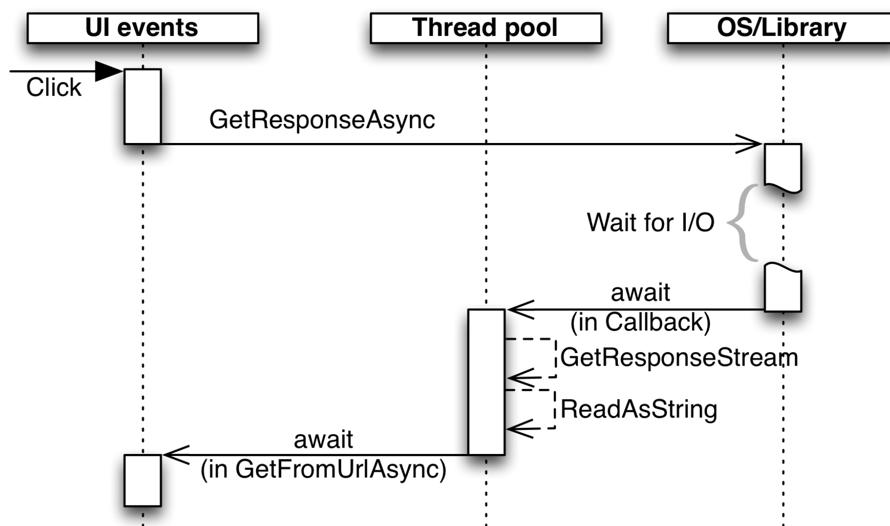

Figure 2.2: Where is the async/await-based code executing?

illustrates its flow of execution.

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. Methods that have the `async` modifier will thus run synchronously up to the first `await` expression (and if it does not have any, it will complete synchronously). Merely adding the `async` modifier does not magically make a method be asynchronously executed in the background.

Where the code is executing. The new `async/await` keywords introduced a new programming paradigm: *pause 'n' play*. 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, which can then continue to do something else. When the *awaited* `Task`'s background operation is completed, the method is resumed from right after the `await` expression. Code 2.8 shows the TAP- & `async/await`-based equivalent of synchronous Code 2.6. Figure 2.2 illustrates its flow of execution.

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 legacy async I/O constructs. Methods that have the `async` modifier will thus run synchronously up to the first `await` expression (and if it does not have any, it will complete synchronously). Merely adding the `async` modifier does not make a method be asynchronously executed in the background.

There is one important difference between `async` continuations, and APM or plain TAP callback continuations: APM and plain TAP always execute the callback on a thread pool thread. The programmer needs to explicitly schedule a UI event to interface with the UI, as shown in Code 2.7 and Figure 2.1.

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. For example, if the `await` keyword is encountered in the UI event thread, it will capture that context. 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 in a sequential manner. See Code 2.8 for an example.

Comparing the code examples in Code 2.6 and Code 2.8 will show that the responsive version based on TAP & `async/await` only slightly differs from the sequential version. It is readable in a similar fashion, and even the UI update (setting contents of the text box) is back at its original place.

ConfigureAwait(false). By default, `await` expressions capture the current context. However, it is not always needed to make the expensive context switch back to the original context. To forestall a context switch, an `await`'ed `Task` can be set to ignore capturing the current context by using `ConfigureAwait(false)`. In Code 2.8, in `GetFromUrlAsync`, none of

the statements following the await expressions require access to the UI. Hence, the `await`'ed `Task` is set with `ConfigureAwait(false)`. In `Button_Click`, the statement following `await GetFromUrlAsync(url)` does need to update the UI. So that `await` expression should capture the original context, and the task should not be set up with `ConfigureAwait(false)`.

2.3 Definitions

We define some terms here that we will frequently use in the dissertation:

async call: asynchronous (non-blocking) method call that returns `Task`.

async I/O call: *async call* that is provided by .NET framework (not user-defined) to asynchronously initiate an I/O operation such as `WebClient.DownloadFileAsync()`.

async method: user-defined method definition that is marked with `async` modifier (see line 1 in Code 2.8).

to wait (verb): synchronously waiting for an *async call* (e.g., `task.Wait()`).

to await (verb): asynchronously waiting for an *async call* (e.g., `await task`).

`await` (noun): a keyword that is inserted in front of an *async call* that you would like to *await* (see line 2 in Code 2.8).

fire & forget async call: *async call* that is neither waited, nor *awaited*.

CHAPTER 3

Migration to Higher-Level Concurrency Constructs

3.1 Introduction

In the quest to support programmers with faster, more scalable, and readable code, parallel libraries continuously evolve from low-level to higher-level abstractions. For example, Java 6 (2006) improved the performance and scalability of its concurrent collections (e.g., `ConcurrentHashMap`), Java 7 (2011) added higher-level abstractions such as lightweight tasks, Java 8 (2014) added lambda expressions that dramatically improve the readability of parallel code. Similarly, in the C# ecosystem, .NET 1.0 (2002) supported a Threading library, .NET 4.0 (2010) added lightweight tasks, declarative parallel queries, and concurrent collections, .NET 4.5 (2012) added reactive asynchronous operations.

Low-level abstractions, such as `Thread`, make parallel code more complex, less scalable, and slower. Because `Thread` represents an actual OS-level thread, developers need to take into account the hardware (e.g., the number of cores) while coding. Threads are *heavyweight*: each OS thread consumes a non-trivial amount of memory, and starting and cleaning up after a retired thread takes hundreds of thousands of CPU cycles. Even though a .NET developer can use `ThreadPool` to amortize the cost of creating and recycling threads, she cannot control the behavior of the computation on `ThreadPool`. Moreover, new platforms such as Microsoft Surface Tablet no longer support `Thread`. .NET also does not allow using the new features (e.g., `async` abstractions) with `Thread` and `ThreadPool`. Furthermore, when developers mix old and new parallel abstractions in their code, it makes it hard to reason about the code because all these abstractions have different scheduling rules.

Higher-level abstractions such as .NET `Task`, a unit of parallel work, make the code less complex. `Task` gives advanced control to the developer (e.g., chaining, cancellation, futures,

callbacks), and is more scalable than `Thread`. Unlike threads, tasks are *lightweight*: they have a much smaller performance overhead and the runtime system automatically balances the workload. Microsoft now encourages developers to use `Task` in order to write scalable, hardware independent, fast, and readable parallel code [23].

However, most developers are oblivious to the benefits brought by the higher-level parallel abstractions. In recent empirical studies for C# [6] and Java [7], researchers found that `Thread` is still the primary choice for most developers. In this study we find similar evidence. Our corpus of the most popular and active 880 C# applications on Github [16] that we prepared for this study, shows that when developers use parallel abstractions they still use the old `Thread` and `ThreadPool` 62% of the time, despite the availability of better options. Therefore, a lot of code needs to be migrated from low-level parallel abstractions to their higher-level equivalents.

The migration has several challenges. First, developers need to be aware of the different nature of the computation. While blocking operations (e.g., I/O operations, `Thread.Sleep`) do not cause a problem in Thread-based code, they can cause a serious performance issue (called thread-starvation) in Task-based code. Because the developers need to search for such operations deep in the call graph of the concurrent abstraction, it is easy to overlook them. For example, in our corpus of 880 C# applications, we found that 32% of tasks have at least one I/O blocking operation and 9% use `Thread.Sleep` that blocks the thread longer than 1 sec. Second, developers need to be aware of differences in handling exceptions, otherwise exceptions become ineffective or can get lost.

In this study, we present an automated migration tool, TASKIFIER, that transforms old style `Thread` and `ThreadPool` abstractions to higher-level `Task` abstractions in C# code. During the migration, TASKIFIER automatically addresses the non-trivial challenges such as transforming blocking to non-blocking operations, and preserving the exception-handling behavior.

The recent versions of parallel libraries provide even higher-level abstractions on top of Tasks. For example, the `Parallel` abstraction in C# supports parallel programming design patterns: data parallelism in the form of parallel loops, and fork-join task parallelism in the form of parallel tasks co-invoked in parallel. These dramatically improve the readability of

the parallel code. Consider the example in Code 3.1, taken from ravendb [28] application. Code 3.2 represents the same code with a `Parallel` operation, which dramatically reduces the code. According to a study [27] by Microsoft, these patterns may also lead to better performance than when using `Task`, especially when there is a large number of work items (`Parallel` reuses tasks at runtime to eliminate the overhead).

Code 3.1 Forking Task in a loop

```
1  List<Task> tasks = new List<Task>();
2  for (int i = 0; i <= n; i++)
3  {
4    int copy = i;
5    Task taskHandle = new Task(
6           () => DoInsert(..., copy));
7    taskHandle.Start();
8    tasks.Add(taskHandle);
9  }
10 Task.WaitAll(tasks);
```

Code 3.2 Equivalent Parallel.For

```
1 Parallel.For(0,n,(i)=>DoInsert(...,i));
```

Despite the advantages of the higher-level abstractions in the `Parallel` class, developers rarely use them. In our corpus we found that only 6% of the applications use the `Parallel` operations. We contacted the developers of 10 applications which heavily use `Thread`, `ThreadPool`, and `Task` abstractions, and asked why they are not using the `Parallel` operations. The major reason given by developers was lack of awareness. This indicates there is a need for tools that suggest transformations, thus educating developers about better coding practices.

Transforming the Task-related code into higher-level `Parallel` operations is not trivial: it requires control- and data-flow analysis, as well as loop-carried dependence analysis. For the example in Code 3.1, the code does not execute the assignment in Line 4 in parallel with itself in other iterations (only the code in the task body – Line 6 – is executed in parallel). However, after converting the original `for` into a `Parallel.For`, the assignment in Line 4 will also execute in parallel with other assignments. Thus, the programmer must reason about the loop-carried dependences.

Inspired from the problems that developers face in practice, we designed and implemented a novel tool, SIMPLIFIER, that extracts and converts Task-related code snippets into higher-

level parallel patterns. To overcome the lack of developer awareness, Simplifier operates in a mode where it suggests transformations as "quick-hints" in the Visual Studio IDE. If the developer agrees with the suggestion, Simplifier automatically transforms the code.

3.2 Formative Study

Before explaining Taskifier and Simplifier, we explore the motivations of these tools by answering two research questions:

RQ1: *What level of parallel abstractions do developers use?*

RQ2: *What do developers think about parallel abstractions?*

We first explain how we gather the code corpus to answer these questions. We use the same code corpus to evaluate our tools (Section 3.5).

3.2.1 Methodology

We created a code corpus of C# apps by using our tool OSSCollector. We chose GitHub [16] as the source of the code corpus because Github is now the most popular open-source software repository, having surpassed Google Code and SourceForge.

OSSCollector downloaded the most popular 1000 C# apps which have been modified at least once since June 2013. OSSCollector visited each project file in apps in order to resolve/install dependencies by using nuget [34], the package manager of choice for apps targeting .NET. OSSCollector also eliminated the apps that do not compile due to missing libraries, incorrect configurations, etc. OSSCollector made as many projects compilable as possible (i.e., by resolving/installing dependencies).

OSSCollector also eliminated 72 apps that targeted old platforms (e.g., Windows Phone 7, .NET Framework 3.5, Silverlight 4) because these old platforms do not support new parallel libraries.

After all, OSSCollector successfully retained 880 apps, comprising 42M SLOC, produced by 1859 developers. This is the corpus that we used in our analysis and evaluation.

In terms of the application domain, the code corpus has (1) 364 libraries or apps for desktops, (2) 185 portable-libraries for cross-platform development, (3) 137 Windows Phone 8 apps, (4) 84 web apps (ASP.NET), (5) 56 tablet applications (Surface WinRT), and (6) 54 Silverlight apps (i.e., client-side runtime environment like Adobe Flash). Hence, the code corpus has apps which (i) span a wide domain and (ii) are developed by different teams with 1859 contributors from a large and varied community.

Roslyn. The Microsoft Visual Studio team has released Roslyn [18] with the goal to expose compiler-as-a-service through APIs to other tools like code generation, analysis, and refactoring. Roslyn has components such as Syntax, Symbol Table, Binding, and Flow Analysis APIs. We used these APIs in our tools for analyzing our code corpus.

Roslyn also provides the Services API allowing to extend Visual Studio. Developers can customize and develop IntelliSense, refactorings, and code formatting features. We used Services API for implementing our tools.

3.2.2 **RQ1:** What level of parallel abstractions do developers use?

In a previous study [6], we found out that developers prefer to use old style threading code over `Task` in C# apps. We wanted to have a newer code corpus which includes the recently updated most popular apps. We used Roslyn API to get the usage statistics of the abstractions.

As we explained in Section 2.2.1, there are 4 main ways to offload a computation to another thread: (1) creating a `Thread`, (2) accessing the `ThreadPool` directly, (3) creating a `Task`, (4) using task or data parallelism patterns with `Parallel.Invoke` and `Parallel.For(Each)`. Table 3.1 tabulates the usage statistics of all these approaches. Some apps use more than one parallel idiom and some never use any parallel idiom.

As we see from the table, developers use `Thread` and `ThreadPool` more than `Task` and `Parallel` even though our code corpus contains recently updated apps which target the latest versions of various platforms. The usage statistics of `Parallel` are also very low compared to `Task`. These findings definitely show that developers use low-level parallel

Table 3.1: Usage of parallel idioms. The three columns show the total number of abstraction instances, the total number of apps with instances of the abstraction, and the percentage of apps with instances of the abstraction.

	#	App	App%
Creating a `Thread`	2105	269	31%
Using `ThreadPool`	1244	191	22%
Creating a `Task`	1542	170	19%
Data Parallelism Pattern with `Parallel.For(Each)`	432	51	6%
Task Parallelism Pattern with `Parallel.Invoke`	53	12	1%

abstractions.

Surprisingly, we also found that 96 apps use `Thread`, `ThreadPool`, and `Task` at the same time. This can easily confuse the developer about the scheduling behavior.

3.2.3 **RQ2:** What do developers think about parallel abstractions?

In this question, we explore why developers use low-level abstractions and whether they are aware of the newer abstractions.

We first asked the experts on parallel programming in C#. We looked for the experts on StackOverflow [35] which is the pioneering Q&A website for programming. We contacted the top 10 users for the tags "multithreading" and "C#", and got replies from 7 of them. Among them are Joe Albahari who is the author of several books on C# (e.g., "C# in a Nutshell"), and John Skeet who is the author of "C# in Depth" and he is regarded as one of the most influential people on StackOverflow.

All of them agree that `Task` should be the only way for parallel and concurrent programming in C#. For example, one said "*Tasks should be the only construct for building multithreaded and asynchronous applications*". According to them, `Thread` should be used for testing purposes: "*threads are actually useful for debugging*" (e.g., guaranteeing a multithreading environment, giving names to threads). When we asked them whether an automated tool is needed to convert `Thread` to `Task`, they concluded that the existence of some challenges makes the automation really hard. For example, one said that "*I wonder whether*

doing it nicely in an automated fashion is even feasible" and another said that "*Often there's in-brain baggage about what the thread is really doing which could affect what the target of the refactoring should actually be*".

Second, we contacted the developers of 10 applications which heavily mix `Thread`, `ThreadPool`, and `Task`. Most of them said that the legacy code uses `Thread` and `ThreadPool` and they always prefer `Task` in the recent code. The developer of the popular ravendb application [28], Oren Eini, said that "*We intend to move most stuff to tasks, but that is on an as needed basis, since the code works*" and another said that his team "*never had time to change them*". This comment indicates that the changes are tedious.

We also asked the developers whether they are aware of the `Parallel` class. Developers of 7 of the apps said that they are not aware of the `Parallel` class and they were surprised seeing how much it decreases the code complexity: "*Is this in .NET framework? It is the most elegant way of a parallel loop*".

3.3 TASKIFIER: Migrating `Thread`-based Code to `Task`

We developed TASKIFIER, a tool that migrates `Thread` and `ThreadPool` abstractions to `Task` abstractions. Section 3.3.1 presents the algorithms for the migration from `Thread` to `Task`. Section 3.3.2 presents the migration from `ThreadPool` to `Task`. Section 3.3.3 presents the special cases to handle some challenges. Section 3.3.4 presents how developers interact with TASKIFIER.

3.3.1 `Thread` to `Task`

First, TASKIFIER needs to identify the `Thread` instances that serve as the target of the transformation. In order to do this, TASKIFIER detects all variable declarations of `Thread` type (this also includes arrays and collections of `Thread`). For each `Thread` variable, it iterates over its method calls (e.g., `thread.Start()`) and member accesses (e.g., `thread.IsAlive=...`). Then, TASKIFIER replaces each of them with their correspondent from the `Task` class. However, corresponding operations do not necessarily use the same name. For instance,

`thread.ThreadState`, an instance field of `Thread` class gets the status of the current thread. The same goal is achieved in `Task` class by using `task.Status`.

Some low-level operations in `Thread` do not have a correspondent in the `Task` class. For example, (1) Priority, (2) Dedicated Name, (3) Apartment State.

After studying both `Thread` and `Task`, we came up with a mapping between them. TASKIFIER uses this map for the conversion. If TASKIFIER finds operations that have no equivalents, it will discard the whole conversion from `Thread` to `Task` for that specific `Thread` variable.

The most important transformations in the uses of `Thread` variables are for creating, starting, and waiting operations. Code 3.3 shows a basic usage of `Thread` and Code 3.4 represents the equivalent code with `Task` operations. Developers create `Thread` by using its constructor and providing the asynchronous computation. There are various ways of specifying the computation in the constructor such as delegates, lambdas, and method names. In the example below, a delegate (`ThreadStart`) is used. TASKIFIER gets the computation from the delegate constructor and transforms it to a lambda function. For starting the `Thread` and `Task`, the operation is the same and for waiting, `Task` uses `Wait` instead of `Join`.

Code 3.3 Simple Thread example

```
1 ThreadStart t = new ThreadStart(doWork);
2 Thread thread = new Thread(t);
3 thread.Start();
4 thread.Join();
```

Code 3.4 Equivalent Task code

```
1 Task task = new Task(()=>doWork());
2 task.Start();
3 task.Wait();
```

While the transformation in Code 3.3 and Code 3.4 show the most basic case when the asynchronous computation does not take any arguments, the transformation is more involved when the computation needs arguments. Consider the example in Code 3.5. The asynchronous computation is the one provided by the `Reset` method (passed in line 1), but the parameter of the `Reset` method is passed as an argument to the `Thread.Start` in line 3. Since the `Thread.Start` can only take `Object` arguments, the developer has to downcast from `Object` to a specific type (in line 7).

Code 3.6 shows the refactored version, that uses `Task`. Unlike in `Thread`, `Task.Start` does not take a parameter. In order to pass the state argument `e` to the asynchronous

computation `Reset`, the code uses a lambda parameter in the `Task` constructor. In this case, since there is no need to cast parameters in the `Reset` method body, TASKIFIER also eliminates the casting statement (Line 7 from Code 3.5).

Code 3.5 Thread with dependent operators from Dynamo[36] app

```
1  ParameterizedThreadStart threadStart = new ParameterizedThreadStart(Reset);
2  Thread workerThread = new Thread(threadStart);
3  workerThread.Start(e);
4  ...
5  private void Reset(object state)
6  {
7    var args = (MouseButtonEventArgs)state;
8    OnClick(this, args);
9    ...
10 }
```

Code 3.6 Code 3.5 migrated to Task

```
1  Task workerTask = new Task(()=>Reset(e));
2  workerTask.Start();
3  ...
4  private void Reset(MouseButtonEventArgs args)
5  {
6    OnClick(this, args);
7    ...
8  }
```

TASKIFIER also changes the variable names such as from `workerThread` to `workerTask` by using the built-in Rename refactoring of Visual Studio.

After TASKIFIER migrates the `Thread` variable to `Task`, it makes an overall pass over the code again to find some optimizations. For instance, in Code 3.6, there is no statement between `Task` constructor and `Start` method. In `Task`, there is a method combining these two statements: `Task.Run` creates a `Task`, starts running it, and returns a reference to it. TASKIFIER replaces the first two lines of Code 3.6 with only one statement:

```
Task workerTask = Task.Run(()=>Reset(e));
```

TASKIFIER successfully detects all variable declarations of `Thread` class type; however, we noticed that developers can use threads through an anonymous instance. The example below from antlrcs app [37] shows such an anonymous usage of `Thread` on the left-hand side, and refactored version with `Task` on the right-hand side. TASKIFIER replaces the `Thread` constructor and the start operation with a static method of `Task`.

```
new Thread(t1.Run).Start(arg);     =>          Task.Run(()=>t1.Run(arg));
```

3.3.2 `ThreadPool` to `Task`

The conversion from `ThreadPool` to `Task` is less complex than the previous transformation. There is only one static method that needs to be replaced, `ThreadPool.QueueUserWorkItem(...)`. TASKIFIER simply replaces this method with the static `Task.Run` method and removes the parameter casting from `Object` to actual type in the beginning of the computation. The example below illustrates the transformation.

```
WaitCallback operation= new WaitCallback(doSendPhoto);
ThreadPool.QueueUserWorkItem(operation, e);
```

```
Task.Run(()=>DoSendPhoto(e));
```

3.3.3 Special cases

There are three special cases that make it non-trivial to migrate from `Thread` and `ThreadPool` to `Task` manually:

I/O or CPU-bound Thread. During manual migration, developers need to understand whether the candidate thread for migration is I/O or CPU bound since it can significantly

affect performance. If an I/O-bound `Thread` is transformed to a `Task` without special consideration, it can cause starvation for other tasks in the thread pool. Some blocking synchronization abstractions like `Thread.Sleep` can also cause starvation when the delay is long.

Manually determining whether the code in a `Thread` transitively calls some blocking operations is non-trivial. It requires deep inter-procedural analysis. When developers convert `Thread` to `Task` manually, it is easy to miss such blocking operations that appear deep inside the methods called indirectly from the body of the `Thread`. In our code corpus, we found that 32% of tasks have at least one I/O blocking operation and 9% use `Thread.Sleep` that blocks the thread longer than 1 second. It shows that developers are not aware of this issue and their tasks can starve.

Thus, it is crucial for TASKIFIER to determine whether the nature of the computation is I/O or CPU-bound. If it finds blocking calls, it converts them into non-blocking calls, in order to avoid starvation.

To do so, TASKIFIER checks each method call in the call graph of the `Thread` body for a blocking I/O operation by using a blacklist approach. For this check, we have the list of all blocking I/O operations in .NET. If TASKIFIER finds a method call to a blocking I/O operation, it tries to find an asynchronous (non-blocking) version of it. For example, if it comes across a `stream.Read()` method call, TASKIFIER checks the members of the `Stream` class to see if there is a corresponding `ReadAsync` method. Upon finding such an equivalent, it gets the same parameters from the blocking version. `ReadAsync` is now non-blocking and returns a future `Task` to get the result when it is available. After finding the corresponding non-blocking operation, TASKIFIER simply replaces the invocation with the new operation and makes it `await`'ed. When a `Task` is awaited in an `await` expression, the current method is paused and control is returned to the caller. The caller is the thread pool so the thread pool will choose another task instead of busy-waiting. When the await'ed `Task`'s background operation is completed, the method is resumed from right after the `await` expression.

```
var string = stream.Read();      =>          var string = await stream.ReadAsync();
```

If TASKIFIER cannot find asynchronous versions for all blocking I/O operations in the `Thread` body, it does not take any risks of blocking the current thread and, instead, it inserts a flag to the `Task` creation statement: `TaskCreationOptions.LongRunning`. This flag forces the creation of a new thread outside the pool. This has the same behavior as the original code, i.e., it explicitly create a new `Thread`. But now the code still enjoys the many other benefits of using Tasks, such as compatibility with the newer libraries and brevity.

In the case of `Thread.Sleep`, TASKIFIER replaces this blocking operation with a timer-based non-blocking version, `await Task.Delay`. Upon seeing this statement, the thread in the thread pool does not continue executing its task and another task from the thread pool is chosen (cooperative-blocking).

Foreground and background Thread. By default, a `Thread` runs in the foreground, whereas threads from `ThreadPool` and `Task` run in the background. Background threads are identical to foreground threads with one exception: a background thread does not keep the managed execution environment running. `Thread` is created on the foreground by default but can be made background by "`thread.IsBackground = true`" statement. If a developer wants to execute `Task` in a foreground thread, she has to add some extra-code in the body of `Task`.

Since the intention is to preserve the original behavior as much as possible, TASKIFIER should do the transformations accordingly. In the example below, the program will not terminate until the method, `LongRunning` reaches the end. However, when this `Thread` is turned into `Task` without any special consideration, the program will not wait for this method and it will immediately terminate. While it is easy to diagnose the problem in this simple example, it can be really hard for a fairly complex app.

```
public static void main(String args[])
{
  ...
  new Thread(LongRunning);
}
```

Although, in some cases, TASKIFIER is able to tell from the context if the thread is foreground or background, it is usually hard to tell if the developer really intended to create

a foreground thread. Developers usually do not put much thought into a thread's being a foreground thread when created. We chose to implement our algorithm for TASKIFIER to transform `Thread` to `Task` by default to work in the background. The developer still has the option of telling TASKIFIER to create foreground tasks; however, the reasoning behind going with the background by default is that when we contacted the developers, most of them did not want the `Task` to work in the foreground even though they created foreground threads.

Exception handling. Another difference between `Thread` and `Task` is the mechanism of unhandled exceptions. An unhandled exception in `Thread` and `ThreadPool` abstractions results in termination of the application. However, unhandled exceptions that are thrown by user code that is running inside `Task` abstractions are propagated back to the joining thread when the static or instance `Task.Wait` methods are used. For a thrown exception to be effective in a `Task`, that `Task` should be waited; otherwise, the exceptions will not cause the termination of the process.

A simple direct migration from `Thread` and `ThreadPool` to `Task` can make the unhanded exceptions silenced so developers will not notice them. This situation may destroy the reliability and error-recovery mechanism that developers put into the original program.

To take care of this, TASKIFIER adds a method call to make sure exception handling is preserved and unhandled exceptions are not ignored when non-waited threads are migrated to tasks. During the transformation of the example below, TASKIFIER adds a new method, `FailFastOnException` to the project just once. Other instances of `Task` in the project can use this method. However, this stage is optional and can be enabled by the user upon request.

```
new Thread(method).Start();
void method()
{
  throw new Exception();
}
```

```
Task.Run(()=>method()).FailFastOnException();
```

```
void method()
{
  throw new Exception();
}
public static Task FailFastOnException(this Task task)
{
    task.ContinueWith(c => Environment.FailFast("Task faulted", c.Exception),
        TaskContinuationOptions.OnlyOnFaulted |
        TaskContinuationOptions.ExecuteSynchronously |
        TaskContinuationOptions.DetachedFromParent);
    return task;
}
```

3.3.4 Workflow

We implemented TASKIFIER as a Visual Studio plugin, on top of the Roslyn SDK [18]. Because developers need to run TASKIFIER only once per migration, TASKIFIER operates in a batch mode. The batch option allows the programmer to migrate automatically by selecting any file or project in the IDE. Before starting the migration, TASKIFIER asks the user for two preferences: *Foreground Thread* option and *Exception Handling* option. When it operates at the file levels, TASKIFIER might still modify other files when necessary (e.g., if the method in `Thread` body is located in another file). TASKIFIER migrates `Thread` and `ThreadPool` abstractions to `Task` in about 10 seconds on an average project (100K SLOC).

3.4 SIMPLIFIER: Refactoring to Parallel Design Patterns

TASKIFIER automatically migrates old-style parallel abstractions (`Thread` and `ThreadPool`) to the modern `Task`. However, there are still some opportunities for higher-level abstractions that can make the code faster and more readable.

`Parallel` class (see Section 2.2.1) provides parallel programming design patterns as a higher-level abstraction over `Task` class. Implementing these design patterns with tasks requires developers to write code with several instances of Tasks. A much simpler alternative is to use a single instance of the `Parallel` class, which encapsulates the main skeleton of

the design patterns. While the direct usage of Tasks affords more flexibility and control, we found out that in many cases, developers do not use the extra flexibility, and their code can be greatly simplified with a higher-level design pattern.

We developed SIMPLIFIER that converts multiple `Task` instances to one of three `Parallel` operations (`Parallel.For`, `Parallel.ForEach`, `Parallel.Invoke`). SIMPLIFIER suggests code snippets that can be transformed to `Parallel` operations and then does the actual transformation on demand. Hence, we divided the explanation of the algorithms into two parts: *Suggestion* and *Transformation*. In the Suggestion part, we explain how SIMPLIFIER chooses the code candidates. In the Transformation part, we explain how SIMPLIFIER transforms these candidates to `Parallel` operations. After explaining the three algorithms, we discuss how developers interact with SIMPLIFIER in Section 3.4.4.

3.4.1 Multiple `Task` instances to `Parallel.Invoke`

SIMPLIFIER offers the transformation of task parallelism pattern composed of a group of `Task` instances to `Parallel.Invoke`. First we explain the properties of code snippets that can be transformed to this operation.

Suggestion. As we explained in Section 2.2.1, `Parallel.Invoke` is a succinct way of creating and starting multiples tasks and waiting for them. Consider the example below. `Parallel.Invoke` code on the right-hand side is the equivalent of the code on the left-hand side. For the purpose of simplifying the code with `Parallel.Invoke`, SIMPLIFIER needs to detect such a pattern before suggesting a transformation.

Code 3.7 Multiple Tasks

```
1 Task t1=new Task(()=>sendMsg(arg1));
2 Task t2=new Task(()=>sendMsg(arg2));
3 t1.Start();
4 t2.Start();
5 Task.WaitAll(t1,t2);
```

Code 3.8 Equivalent with Invoke

```
1 Parallel.Invoke(()=>sendMsg(arg1),
2                 ()=>sendMsg(arg2));
```

Code 3.7 shows the simplest form of many variations of code snippets. In order to find as many fits as possible, we need to relax and expand this pattern to detect candidates.

First step to detect the pattern is that the number of `Task` variables should be at least 2, as `Parallel.Invoke` can take unlimited work items as parameters. Second, SIMPLIFIER has to consider that there are many syntactic variations of task creation and task starting operations. Also, there are some operations that combine both *creation* and *starting* like `Task.Factory.StartNew` and `Task.Run` methods. Third, one should keep in mind that there is no need to separate the creation of Tasks into one phase and starting them into another. Each Task can be created then started immediately. Fourth, there may be other statements executing concurrently in between the start of a `Task` and the barrier instruction that waits for all spawned tasks. In case of such statements, SIMPLIFIER encapsulates them in another `Task` and passes the task to `Parallel.Invoke`. Code 3.9 shows a more complex pattern of task parallelism from a real-world app and demonstrates the last point.

After SIMPLIFIER finds out the code snippets that fit into the pattern stated above, it checks if some preconditions hold true to ensure that the transformation is safe. These preconditions are not limitations of SIMPLIFIER; they are caused by how `Parallel.Invoke` encapsulates the task parallelism pattern. Because it is a higher-level abstraction, it waives some advanced features of `Task`. The preconditions are:

P1: None of the `Task` variables in the pattern can be result-bearing computations, i.e., a *future* – `Task<ResultType>` – also called a *promise* in C#. The reason is that after the transformation, there is no way to access the result-bearing from the `Parallel` class.

P2: There should be no reference to the `Task` variables outside of code snippet of the design pattern. Such references will no longer bind to a `Task` after the transformation eliminates the `Task` instances.

P3: None of the `Task` variables in the pattern can use the chaining operation (`ContinueWith`). Since the chaining requires access to the original task, this task will no longer exist after the transformation.

Transformation. If SIMPLIFIER finds a good match of code snippets, its suggestion can be executed and turned into a transformation which yields `Parallel.Invoke` code. Code 3.10 shows the code after the transformation of Code 3.9.

During transformation, the main operation is to get work items from `Task` variables. In the example below, the work item of first `Task` is `()=> DoClone(...)`. These work items can be in different forms such as method identifiers, delegates, or lambda functions as in the example below. SIMPLIFIER handles this variety of forms by transforming the work items to lambda functions.

After SIMPLIFIER gets the work items for the tasks `t1` and `t2`, it forms another work item to encapsulate the statements between task creation and task waiting statements (line 3 and 4 in Code 3.9).

SIMPLIFIER gives all these work items in the form of lambda functions to `Parallel.Invoke` method as parameters. It replaces the original lower-level task parallelism statements with this `Parallel.Invoke` method.

Code 3.9 Candidate from Kudu[38] app

```
1 var t1 = Task.Factory.StartNew(() => DoClone("PClone1", appManager));
2 var t2 = Task.Factory.StartNew(() => DoClone("PClone2", appManager));
3 ParseTheManager();
4 DoClone("PClone3", appManager);
5 Task.WaitAll(t1, t2);
```

Code 3.10 Equivalent Parallel.Invoke code

```
1 Parallel.Invoke(() => DoClone("PClone1", appManager),
2                 () => DoClone("PClone2", appManager),
3                 () => {ParseTheManager();
4                        DoClone("PClone3", appManager);})
```

3.4.2 Fork join tasks to `Parallel.For`

SIMPLIFIER can transform a specific data parallelism pattern to `Parallel.For`. First we explain the properties of code snippets that can be transformed to this operation.

Suggestion. As we explained in Section 2.2.1, `Parallel.For` is a more concise way to

express the pattern of forking several tasks and then waiting for them all to finish at a global barrier.

Considering the example below, the `Parallel.For` code on the right is the equivalent of the code on the left.

Code 3.11 Forking tasks in a loop

```
1 Task[] tasks = new Task [n];
2 for(int i=0; i<n; i++)
3 {
4   int temp = i;
5   tasks[i]= new Task(
6                ()=>Queues[temp].Stop());
7   tasks[i].Start();
8 }
9 Task.WaitAll(tasks);
```

Code 3.12 Equivalent with Parallel.For

```
1 Parallel.For(0,n,(i)=> Queues[i].Stop());
```

Simplifier needs to detect usages of Tasks that form the pattern on the left example above. The code snippet in Code 3.11 is one of the basic representatives of this design pattern; there are other variations who fit the pattern. First thing the tool looks for in the code to decide if it matches the pattern is that the increment operation of the loop must be of the form `++` or `+= 1` (i.e., increments should only be by 1). The loop boundaries do not matter as long as they are integers. Second, as explained in Section 3.4.1, there may be many syntactic variations for the task creation and starting operations.

Third, the collection of tasks does not have to be of type `Array`, they may be of another type like `List`. In this case, tasks are added with `tasks.add(...)` method to the collection in the loop. Fourth, as long as there is no modification to the collection, there may be other statements between creating the collection of tasks and the for loop. During the transformation, these statements are not discarded and they take place before `Parallel.For`.

Fifth, there might be other statements in the loop besides task creation, starting, and adding to the collection. In the Code 3.11 above, there is one such statement: `int temp=i;`. This causes each task to have its own copy of the loop index variable during the iteration of the loop.

Sixth and last, some simple assignment operations may also exist between the loop and the barrier operation that waits for all spawned tasks. Code 3.13 shows a more complex

pattern of data parallelism from a real-world app and demonstrates the last point with the statements in Line 8-9.

After SIMPLIFIER detects the code snippets that fit into the pattern stated above, it checks some preconditions ensuring that the transformation is safe. These preconditions are the result of how the `Parallel.For` encapsulates the date parallelism pattern.

P1, P2, P3: The first three preconditions are the same as the first three preconditions in Section 3.4.1.

P4: The operations in the loop except the task-related statements should not carry any dependence between iterations. Consider the Code 3.13, the statements in Line 4-5 will sequentially execute because they are not included in `Task`. After transforming to `Parallel.For`, the whole body of the loop will be parallelized.

P5: The statements after the loop (e.g., Line 8-9 in Code 3.13) but before the `Task.WaitAll` should not access any data from the Task body. At first, Simplifier did not allow any statement between the loop and `Task.WaitAll`. After we manually analyzed the statements between the loop and `WaitAll` in our code corpus, we noticed that many of them are simple variable declarations which do not use any data from the loop and do not contain any method call sites like in the Code 3.13. Therefore, we relaxed this precondition and allowed the statements after the loop but before the `Task.WaitAll` unless they do not access any data from the `Task` body in the loop. To detect such cases, SIMPLIFIER used an intra-procedural data-flow analysis to determine that these statements are independent from the loop. Roslyn [18] provides ready-to-use control & data flow analysis APIs that SIMPLIFIER used to understand how variables flow in and out of regions of source.

Transformation. Code 3.14 shows the code after the transformation of Code 3.13 showing a more complex example.

During transformation, the main operation is to the get loop boundaries and the work item from the task in the loop. In the example below (Line 6), the work item is `()=>`

`MultiSearcherCallableNoSort(...)`. The loop boundaries are 0 and `tasks.Length`. However, the collection of tasks will be deleted after the transformation. Hence, when SIMPLIFIER detects such a dependence on the size of task collections in the loop boundaries, it replaces this boundary with the original size of the task collection, which is `searchables.Length`.

Then, SIMPLIFIER needs to make sure that the statements in the loop (e.g. Line 4-5 in Code 3.13) are not dependent on loop iterations. If they are not, these statements are put in the beginning of the work item; otherwise, the transformation will not occur. If one of these statements is the temporary holder of the iteration value like `cur = i` in the example below, SIMPLIFIER removes it and replaces the holder (`cur`) with the iteration variable (`i`) in the work item as seen in Code 3.14.

Lastly, SIMPLIFIER replaces the original lower-level data parallelism statements with the `Parallel.For` method.

Code 3.13 Candidate from lucene.net[39] app

```
1 Task[] tasks = new Task[searchables.Length];
2 for (int i = 0; i < tasks.Length; i++)
3 {
4   int cur = i;
5   cur = callableIterate(cur);
6   tasks[i] = Task.Factory.StartNew(() => MultiSearcherCallableNoSort(cur, ...));
7 }
8 int totalHits = 0;
9 float maxScore = float.NegativeInfinity;
10 Task.WaitAll(tasks);
```

Code 3.14 Equivalent Parallel.For code

```
1 Parallel.For(0, searchables.Length, (i) => {
2                             i = callableIterate(i);
3                             MultiSearcherCallableNoSort(i, ...);} );
4 int totalHits = 0;
5 float maxScore = float.NegativeInfinity;
```

3.4.3 Fork join tasks to `Parallel.ForEach`

While this transformation is very similar to `Parallel.For`, it transforms `foreach` loops instead of `for` loops. `foreach` loops are a special case of loops that are used to iterate over the elements of a collection.

First we explain the properties of code snippets that can be transformed to this operation.

Suggestion. Considering the example below, the `Parallel.ForEach` code on the right is the equivalent of the code on the left.

Code 3.15 Equivalent Task example

```
1 Task[] tasks = new Task[sables.Length];
2 foreach (var sable in sables)
3 {
4   tasks[i] = Task.Run(
5            () => sable.DocFreq(term));
6 }
7 Task.WaitAll(tasks);
```

Code 3.16 Parallel.ForEach example

```
1 Parallel.ForEach(sables,
2            (sable)=>sable.DocFreq(term));
```

Simplifier needs to detect usages of Tasks in a `foreach` loop that form the pattern on the left example above. We will generalize this pattern with the same 5 variations in the `Parallel.For` algorithm, except the first one which represents the custom loop boundaries.

After Simplifier detects the code snippets that fit into the pattern, it checks for the same preconditions as in the `Parallel.For` transformation.

Transformation. Code 3.18 shows the code after the transformation of Code 3.17. The transformation is done in a very similar manner with the `Parallel.For` version, except the loop boundaries.

After the work item is extracted from `Task`, Simplifier needs to get the collection variable and iteration variable from the loop declaration (`functions`, `functionText`). Then, Simplifier replaces the original lower-level data parallelism statements with the `Parallel.ForEach` method.

Code 3.17 Candidate from Jace[40] app

```
1 List<Task> tasks = new List<Task>();
2 foreach (string functionText in functions)
```

```
3 {
4   Task task = new Task(() =>
5   {...
6     function(functionText, ...); ...
7   });
8   tasks.Add(task);
9   task.Start();
10 }
11 Task.WaitAll(tasks.ToArray());
```

Code 3.18 Equivalent Parallel.ForEach code

```
1 Parallel.ForEach(functions,(functionText) =>{...
2                                               function(functionText, ...); ...
3                                             });
```

3.4.4 Workflow

We implemented SIMPLIFIER as a Visual Studio plugin, on top of the Roslyn SDK [18]. SIMPLIFIER's workflow is similar to a "quick hint" option which exists in major IDEs such as Eclipse, Netbeans, IntelliJ. SIMPLIFIER scans the file that is open in the editor in real-time. It tries to find code snippets that fit into the patterns of the three transformations discussed above. Because it executes on the background (triggered by any keystroke), the analysis of finding code snippets should be fast enough to prevent sluggishness. However, the analyses for `Parallel.For(Each)` require some expensive checking of preconditions such as P4 and P5 in Section 3.4.2. Because they require dependence and data-flow analyses, we do not execute them in the suggestion phase, but in the transformation phase.

If SIMPLIFIER finds candidates, it suggests the places where the transformations can be useful by underlining the code snippet and displaying a hint in the sidebar. After the user clicks the hint and confirms, SIMPLIFIER transforms the code for the `Parallel.Invoke`. SIMPLIFIER tests long-running preconditions, such as for the `Parallel.For(Each)`, in the transformation phase. If the candidate passes these preconditions too, the code will be transformed to the `Parallel.For(Each)`. If not, SIMPLIFIER will give an informative warning.

3.5 Evaluation

We conducted two kinds of empirical evaluation. First, we *quantitatively* evaluate based on case studies of using our tools on open-source software. Second, we *qualitatively* evaluate based on patches that we sent to open-source developers.

3.5.1 Quantitative

To quantitatively evaluate the usefulness of TASKIFIER and SIMPLIFIER, we answer the following research questions:

EQ1: *How **applicable** are the tools?*

EQ2: *Do the tools reduce the **code bloat**?*

EQ3: *How much **programmer effort** is saved by the tools?*

EQ4: *Are the automated transformations **safe**?*

Experimental setup. To answer the questions above, we ran TASKIFIER and SIMPLIFIER on our code corpus that we gathered from Github. The code corpus has 880 C# apps, comprising 42M SLOC, spanning a wide spectrum from web & desktop apps to libraries and mobile apps.

We ran both tools in batch mode over this code corpus. Even though SIMPLIFIER was not designed to run in a batch mode, we implemented a batch mode specifically for the purpose of the evaluation. TASKIFIER visits all `Thread` variable declarations and anonymous instances, and applies the migration algorithm. SIMPLIFIER finds the candidates of code snippets for each source file, then transforms the snippets to the targeted pattern.

Table 3.2 summarizes the results for the first three research questions.

EQ1: How applicable are the tools? Out of our corpus of 880 apps, 269 used Threads (see Table 3.1). Together, they account for 2105 `Thread` instances. Based on our discussion with experts (see Section 3.2.3), they suggested we discard `Thread` usages in test code because

Table 3.2: Applicability of TASKIFIER and SIMPLIFIER. The first column shows the total number of instances that the tool applied. The second column shows the total number of instances that the tool successfully converted and the third column shows the percentage of successfully transformed instances. The fourth column shows the total number of reduced SLOC by the transformations and the fifth column shows the percentage of the reduced lines. The last column shows the total number of modified SLOC.

	Applicability			Reduction		Modified
	Applied	Conv.	Conv. %	SLOC	%.	SLOC
`Thread` to `Task`	1782	1390	78%	2244	24%	8876
`ThreadPool` to `Task`	1244	1244	100%	173	14%	2115
`Task` to `Parallel.Invoke`	85	85	100%	502	44%	1870
`Task` to `Parallel.For(Each)`	205	188	92%	1918	62%	5640

developers may need threads for enforcing a multithreading testing environment. After eliminating the `Thread` usages in test code, we were left with 1782 `Thread` instances in production code, as shown in Table 3.2.

TASKIFIER migrated 78% of the `Thread` instances. The remaining 22% of `Thread` instances used operations that are not available in the `Task` class, thus are not amenable for migration. For example, one can set up the name of a Thread, but not of a Task. Deciding whether the name is important requires domain knowledge, thus Taskifier stays on the safe side and warns the programmer.

Because there are no preconditions for the migration of `ThreadPool` instances, TASKIFIER migrated all of them to `Task`.

As for SIMPLIFIER, it successfully transformed 100% of the 85 `Task`-based fork-join patterns to `Parallel.Invoke`. Out of the 205 identified `Task`-based data-parallelism patterns, it transformed 92% to `Parallel.For` or `Parallel.ForEach`. The remaining 8% did not pass the preconditions. A major number of them was failed due to P4: loop-carried dependence.

EQ2: Do the tools reduce the code bloat? The second column, *Reduction*, of Table 3.2 shows by how much each tool eliminates bloated code. As we expect, because SIMPLIFIER transforms *multiple* `Task` operations and helper operations to *one* equivalent method in the `Parallel` class (i.e., a *many-to-one* transformation), it has the largest impact. For the trans-

formation to `Parallel.Invoke`, SIMPLIFIER achieved on average a 44% reduction in SLOC for each code snippet that it transformed. For the transformation to `Parallel.For(Each)`, it achieved on average a 62% reduction for each transformed code snippet.

TASKIFIER migrates *one* `Thread` operation to *one* equivalent `Task` operation (i.e., a *one-to-one* transformation), so we expect modest reductions in LOC. These come from optimizations such as combining the creation and start `Task` operations, removing explicit casting statements which are not needed in `Task` bodies, etc. However, the advantages brought by TASKIFIER are (i) the modernization of the legacy code so that it can now be used with the newer platforms, and (ii) the transformation of blocking operations to the equivalent non-blocking operations.

EQ3: How much programmer effort is saved by the tools? The last column of Table 3.2 shows that the transformations are tedious. Had the programmers manually changed the code, they would have had to manually modify 10991 SLOC for the migration to `Task` and 7510 SLOC for the migration to `Parallel`.

Moreover, these changes are non-trivial. TASKIFIER found that 37% of `Thread` instances had at least one I/O blocking operation. To find these I/O blocking operations, TASKIFIER had to check deeper in the call-graphs of `Thread` bodies, which span 3.4 files on average. SIMPLIFIER found that 42% of the loops it tried to transform contained statements that needed an analysis to identify loop-carried dependences.

EQ4: Are the automated transformations safe? We used two means to check the safety of our transformations. First, after our tools applied any transformation, our evaluation script compiled the app in-memory and determined that no compilation errors were introduced. Second, we sampled and manually checked 10% of all transformed instances and determined that they were correct. Also, the original developers of the source code thought that the transformations were correct (see Section 3.5.2).

In contrast to the code that was transformed with the tools, we found that 32% of the `Task`-code manually written by open-source developers contained at least one I/O blocking operation which can cause serious performance issues (see Section 3.3.3). However, the code transformed by TASKIFIER into `Task` instances does not have this problem.

3.5.2 Qualitative evaluation

To further evaluate the usefulness our of tools in practice, we identified actively developed C# applications, we ran our tools on them, and submitted patches to the developers.

For Taskifier, we selected the 10 most recently updated apps that use `Thread` and `ThreadPool` and transformed them with Taskifier. We submitted 52 patches via a pull request. Developers of 8 apps out of 10 responded, and accepted 42 patches.

We received very positive feedback on these pull requests. Some developers said that migration to `Task` is on their TODO list but they always postponed it because of working on new features. It is tedious to migrate `Task` and developers can easily miss some important issues such as blocking I/O operations during the migration. Taskifier helps them migrate their code in a fast and safe manner.

For Simplifier, we selected a different set of 10 most recently updated apps that had a high chance of including good matches of code snippets for `Parallel.For(Each)` or `Parallel.Invoke` patterns. We submitted 14 patches. Developers of 7 apps out of 10 responded, and accepted 11 patches. All of them liked the new code after the transformation and asked us whether we can make the tool available now.

3.5.3 Discussion

As explained in Section 3.3.3, Taskifier analyzed the call graph of `Thread` body to detect I/O blocking operations, using a blacklist approach. Although we have the list of I/O blocking operations in .NET framework, Taskifier is not aware of I/O blocking operations implemented by 3rd-party libraries whose source code is not available in the app. However, we don't expect that the number of blocking I/O operations implemented by external libraries to be high.

Most of non-blocking I/O and synchronization operations were released in .NET 4.5 (2012). If an application does not target .NET 4.5, it cannot take advantage of the non-blocking operations. However, applications that are targeting the new platforms (e.g, Windows Phone 8, Surface) are forced to use .NET 4.5.

With respect to releasing Taskifier and Simplifier, we will be able to publish the

tools when Microsoft publicly releases the new version of Roslyn (expected by Spring '14). Because we used an internal version of Roslyn, we had to sign an NDA, which prohibits us from releasing tools based on Roslyn.

3.6 Summary

To make existing parallel code readable, faster, and scalable, it is essential to use higher-level parallel abstractions. Their usage is encouraged by the industry leaders as the old, low-level abstractions are subject to deprecation and removal in new platforms.

Our motivational study of a corpus of 880 C# applications revealed that many developers still use the lower-level parallel abstractions and some are not even aware of the better abstractions. This suggests a new workflow for transformation tools, where suggestions can make developers aware of new abstractions.

Converting from low-level to high-level abstractions cannot be done by a simple find-and-replace tool, but it requires custom program analysis and transformation. For example, 37% of `Thread` instances use blocking I/O operations, which need special treatment when they are converted to `Task` instances, otherwise it can create severe performance bugs. We found that 32% instances of manually written `Task` indeed contain blocking I/O operations.

In this study we presented two tools. Our first tool, TASKIFIER, converts `Thread`-based usage to lightweight `Task`. We were surprised that despite some differences between `Thread` and `Task` abstractions, 78% of the code that uses `Thread` can be successfully converted to `Task`. Our second tool, SIMPLIFIER, converts `Task`-based code into higher-level parallel design patterns. Such conversions reduce the code bloat by 57%. The developers of the open-source projects accepted 53 of our patches and are looking forward to using our tools.

CHAPTER 4

Refactoring to `async/await` Keywords

4.1 Introduction

User interfaces are usually designed around the use of a single user interface (UI) event thread [30, 31, 32, 33]: every operation that modifies UI state is executed as an event on that thread. The UI "freezes" when it cannot respond to input, or when it cannot be redrawn. It is recommended that long-running CPU-bound or blocking I/O operations execute asynchronously so that the application (app) continues to respond to UI events. Asynchronous programming is in demand today because responsiveness is increasingly important on all modern devices: desktop, mobile, or web apps. Therefore, major programming languages have APIs that support non-blocking, asynchronous operations (e.g., to access the web, or for file operations). While these APIs make asynchronous programming possible, they do not make it easy.

Asynchronous APIs rely on callbacks. However, callbacks invert the control flow, are awkward, and obfuscate the intent of the original synchronous code [9].

Recently, major languages (F# [9], C# and Visual Basic [10], and Scala [11]) introduced `async` constructs that resemble the straightforward coding style of traditional synchronous code. Thus, they recognize asynchronous programming as a first-class citizen.

Yet, we know little about how developers use asynchronous programming and specifically the new `async` constructs in practice. Without such knowledge, other developers cannot educate themselves about the state of the practice, language and library designers are unaware of any misuse, researchers make wrong assumptions, and tool providers do not provide the tools that developers really need. This knowledge is also important as a guide to designers of other major languages (e.g., Java) planning to support similar constructs. Hence, asyn-

chronous programming deserves first-class citizenship in empirical research and tool support, too.

We present the first study that analyzes the usage of asynchronous libraries and new language constructs, `async/await` in both industry and open-source communities. We analyze 1378 open-source Windows Phone (WP) apps, comprising 12M SLOC (C#), produced by 3376 developers. In industry, we analyze 3.2M SLOC, comprising various platforms written in C# (console, web, and library) in two companies.

We focus on WP platform in the open-source apps because we expect to find many exemplars of asynchronous programming, given that responsiveness is critical. Mobile apps can easily be unresponsive because mobile devices have limited resources and have high latency (excessive network accesses). With the immediacy of touch-based UIs, even small hiccups in responsiveness are more obvious and jarring than when using a mouse or keyboard. Some sluggishness might motivate the user to uninstall the app, and possibly submit negative comments in the app store [41]. Moreover, mobile apps are becoming increasingly more important. According to Gartner, by 2016 more than 300 billion apps will be downloaded annually [42].

The goal of this study is twofold. First, we obtain a deep understanding of the problems around asynchronous programming. Second, we present a toolkit (two tools) to address exactly these problems. To this end, we investigate both industry and open-source codebases through tools and by hand, focussing on the following research questions:

RQ1: *How do developers use asynchronous programming?*

RQ2: *To what extent do developers misuse* `async`*?*

We found that developers heavily use callback-based asynchronous idioms. However, Microsoft officially no longer recommends these asynchronous idioms [12] and has started to replace them with new idioms in new libraries (e.g., WinRT). Developers need to refactor callback-based idioms to new idioms that can take advantage of the `async/await` keywords. The changes that the refactoring requires are non-trivial, though. For instance, developers have to inspect deep call graphs. Furthermore, they need to be extra careful to preserve

exception handling behavior. Thus, we implemented the refactoring as an automated tool, Asyncifier.

We also found that nearly half of open-source WP8 apps have started to use the 9-month-old `async/await` keywords. However, developers frequently misuse `async/await` in both industry and open-source codebases. We define *misuse* as anti-patterns, which hurt performance and might cause serious problems like deadlocks. For instance, we found that 14% of methods that use (the expensive) `async/await` keywords do this unnecessarily, 19% of methods do not follow an important good practice [14], 1 out of 5 apps misses opportunities in async methods to increase asynchronicity, and developers (almost) always unnecessarily capture context, hurting performance. Thus, we implemented a transformation tool, AsyncFixer, that finds and corrects the misused `async`.

4.2 Formative Study

We are interested in assessing the usage of state of the art asynchronous programming in real world apps from industry and open-source codebases.

4.2.1 Methodology

Open-source codebase. We chose Microsoft's CodePlex [43] and GitHub [16] as sources of the code corpus of WP apps. According to a recent study [44], most C# apps reside in these two repositories. We developed OSSCollector to create our code corpus.

We used OSSCollector to download all recently updated WP apps which have a WP-related signature in their project files. It ignores (1) apps without commits since 2012, and (2) apps with less than 500 non-comment, non-blank lines of code (SLOC). The latter "toy apps" are not representative of production code.

OSSCollector makes as many projects compilable as possible (e.g., by resolving-installing dependencies), because the Roslyn APIs that we rely on (see Analysis Infrastructure) require compilable source code.

OSSCollector successfully downloaded and prepared 1378 apps, comprising 12M SLOC, produced by 3376 developers. Our analysis uses all apps, without sampling. OSSCollector just downloaded the version from the main development branch as of August 1st, 2013 for each app.

In our corpus, 1115 apps are targeting WP7, released in October 2010. Another 349 apps target WP8, released in October 2012. 86 apps target both platforms.

Proprietary codebase. To gather the industry data, we had extensive visits at two companies. We first ran our analysis tools on the whole C# codebase of a leading company that provides health information services. We analyzed 405 compilable projects that target library, console, and ASP.NET (web) platforms, comprising 2.5M SLOC. We did not include any test projects in our analyses. Our analysis was applied to the codebase from the main development branch as of October 1st, 2014.

As a second opportunity for the industry data, we analyzed the C# codebase for a new build language which is being developed by a leading technology company. We analyzed 0.6M SLOC that target library and console platforms from the main development branch as of August 1st, 2015.

In total, we analyzed 3.2M SLOC to gather statistics for the industry data.

Analysis infrastructure. We developed AsyncAnalyzer to perform the static analysis of asynchronous programming construct usage. We used Microsoft's recently released Roslyn [18] SDK, which provides an API for syntactic and semantic program analysis, AST transformations, and editor services in Visual Studio.

We executed AsyncAnalyzer in our corpus. We developed a specific analysis to answer each research question.

4.3 How do Developers Use Asynchronous Programming?

We analyzed the usage of both APIs and language keywords for asynchronous programming.

4.3.1 Asynchronous APIs

We detected all APM and TAP methods that are used as async I/O operations in our code corpus as shown in Tables 4.1 and 4.2.

We also detected the ways of executing CPU-bound operations asynchronously. There are two ways to offload CPU-bound operations to another thread: by creating a new thread, or by reusing threads from the thread pool. Based on C# books and references [46], we distinguish 4 different approaches developers use to access the thread pool: (1) the `BackgroundWorker` class, (2) accessing the `ThreadPool` directly, (3) creating `Task`s, and (4) implicitly creating `Task`s by using `Parallel` class. Tables 4.1 and 4.2 tabulate the usage statistics of all these approaches.

Open-source corpus. Because in WP7 apps, TAP methods are only accessible via additional libraries, Table 4.2 tabulates the usage statistics for WP7 and WP8 apps separately. The data shows that APM is more popular than TAP for both WP7 and WP8.

Because `Task` is only available in WP7 apps by using additional libraries, the table shows separate statistics for WP7 and WP8 apps. The data shows that `Task` is used significantly more in WP8 apps, most likely because of availability in the core platform.

We also manually inspected all APM and TAP methods used and categorized them based on the type of I/O operations (see Table 4.3). We found that asynchronous operations are most commonly used for network operations (see Table 4.4).

Proprietary corpus. Surprisingly, the industry codebases use new TAP methods much more than old-fashioned APM methods. To offload CPU-bound operations, the codebase rarely uses old APIs such as `BackgroundWorker`, `ThreadPool`, and `Thread` operations. New `Task` and `Parallel` APIs are preferred.

4.3.2 Missing opportunities for asynchronous I/O

To make the application more scalable, developers should use `async` I/O operations because synchronous I/O operations are going to waste CPU resources by blocking threads that consume non-trivial amount of memory. Therefore, we detected the locations of blocking

Table 4.1: Usage of asynchronous idioms in the proprietary code. The column per codebase shows the total number of idiom instances.

Type		Industry #1	Industry #2
		# callsites	# callsites
I/O-bound	APM Methods	0	42
	TAP Methods	189	101
CPU-bound	New Thread	17	15
	BG Worker	0	5
	ThreadPool	0	2
	New Task	119	90
	Parallel.For/Each	42	72
	Parallel.Invoke	6	0

Table 4.2: Usage of asynchronous idioms in the open-source code. The three columns per codebase show the total number of idiom instances, the total number of apps with instances of the idiom, and the percentage of apps with instances of the idiom.

Type		Open-source WP7			Open-source WP8		
		# callsites	# apps	% apps	# callsites	# apps	% apps
I/O bound	APM Methods	1028	242	22%	217	65	19%
	TAP Methods	123	23	2%	269	57	16%
CPU bound	New Thread	183	92	8%	28	24	7%
	BG Worker	149	73	6%	11	6	2%
	ThreadPool	386	103	9%	52	24	7%
	New Task	51	11	1%	182	28	8%
	Parallel.For/Each	10	5	0.5%	45	22	6%
	Parallel.Invoke	6	3	0.3%	19	10	3%

Table 4.3: Most popular I/O operations used in the open-source code

	Method	#
APM	WebRequest.BeginGetResponse	725
	WebRequest.BeginGetRequestStream	268
	Stream.BeginRead	84
	Stream.BeginWrite	66
	CommittableTransaction.BeginCommit	60
TAP	HttpContent.ReadAsStringAsync	45
	OpenStreamForReadAsync	40
	Stream.CopyToAsync	32
	OpenStreamForWriteAsync	30
	ReadToEndAsync	25

Table 4.4: Catalog of I/O operations used in the open-source code

Type	# Call Sites
Network	1012
File System	310
Database	145
User Interaction	102
Other I/O (speech recog.)	68

I/O operations which should be replaced with corresponding async versions. To detect these operations, AsyncAnalyzer looks up symbols of each method invocation in the codebases. After getting symbol information, the tool looks at the other members of the containing class of that symbol to check whether there is an asynchronous version. For instance, if there is an `x.Read()` method invocation and `x` is an instance of the `Stream` class, the tool looks at the members of the `Stream` class to see whether there is a `ReadAsync` method that gets the same parameters and returns `Task`.

However, it is not necessary and efficient to replace every synchronous I/O operation with their corresponding `async` operation. For relatively fast I/O operations, the overhead of

async I/O requests may make asynchronous I/O less beneficial, particularly if many fast I/O operations need to be made (e.g., in a loop). In this case, synchronous I/O would be better for fast local I/O operations. Because of the possibility of very long timeouts, network I/O should always be async, even if it is to only send 1 single byte.

Because finding fast local I/O operations requires dynamic analysis and depends on the input size, we could not filter out these fast operations. It is still valuable to have a list of potentially blocking I/O operations because the locations of these operations are great places to use a profiler and detect the bottlenecks.

Open-source corpus. We found 19 blocking I/O operations per app on average.

Proprietary corpus. We detected 4512 blocking I/O operations in total (11 per C# project on average).

4.3.3 Language constructs (async/await)

Open-source corpus. async/await have become accessible for WP development in last quarter of 2012. While they are available by default in WP8, WP7 apps have to reference Microsoft.Bcl.Async library to use them.

We found that 45% (157) of WP8 apps use async/await keywords. While nearly half of all WP8 apps have started to use the new 9-month-old constructs, only 10 WP7 apps use them. In the combined 167 apps, we found that there are 2383 async methods that use at least one await keyword in their method body. An async method has 1.6 await keywords on average, meaning that async methods call other async methods.

Proprietary corpus. We found 545 async methods from two industry codebases.

Callback-based APM is the most widely used idiom. While nearly half of all WP8 apps have started to use async/await, only 10 WP7 apps use them.

4.4 Do Developers Misuse `async/await`?

Because `async/await` are relatively new language constructs, we have also investigated how developers misuse them. We define misuse as anti-patterns which hurt performance and might cause serious problems like deadlocks. We detected the following typical misusage idioms.

4.4.1 Fire & forget methods

The `async/await` methods can return `void` instead of `Task`. This means they are "fire&forget" methods, which cannot be awaited. Exceptions thrown in such methods cannot be caught in the calling method, and cause termination of the app. Unless these methods are UI event handlers, this is a code smell. Such methods should return `Task`, which does not force the developer to change anything else, but it does enable easier error handling, composition and testability.

Open-source corpus. 799 of 2382 `async/await` methods are "fire&forget" methods. However, we found that only 339 out of these 799 `async void` methods are event handlers. It means that 19% of all `async` methods (460 out of 2383) are not following this important practice [14]. **One in five async methods violate the principle that an `async` method should be awaitable unless it is the top level event handler.**

Proprietary corpus. Only 10 of 545 `async` methods return `void`. However, all of these 10 `async void` methods are not event handlers so they need to return `Task` instead of `void`.

4.4.2 Unnecessary `async/await` methods

Consider the example from "Cimbalino Windows Phone Toolkit" [47]:

```
public async Task<Stream> OpenFileForReadAsync(...) {
  return await Storage.OpenStreamForReadAsync(path);
}
```

The `OpenStream` method is a TAP call, which is awaited in the `OpenFile` method. However,

there is no need to await it. Because there is no statement after the `await` expression except for the return, the method is paused without reason: the `Task` that is returned by `Storage.OpenStream` can be immediately returned to the caller. The snippet below behaves exactly the same as the one above:

```
public Task<Stream> OpenFileForReadAsync(...) {
  return Storage.OpenStreamForReadAsync(path);
}
```

It is important to detect this kind of misuse. Adding the `async` modifier comes at a price: the compiler generates some code in every `async` method and generated code complicates the control flow which results in decreased performance.

Open-source corpus. We discovered that in 26% of the 167 apps, 324 out of all 2383 `async` methods unnecessarily use `async`. **There is no need to use `async/await` in 14% of `async` methods.**

Proprietary corpus. 35 of 545 `async` methods unnecessarily use `async/await` keywords.

4.4.3 Using long-running operations under `async` methods

We also noticed that developers use some potentially long-running operations under `async` methods even though there are corresponding asynchronous versions of these methods in .NET or third-party libraries. Consider the following example from indulged-flickr [48]:

```
public async void GetPhotoStreamAsync(...) {
  var response = await DispatchRequest(...);
  using (StreamReader reader = new StreamReader(...)){
    string jsonString = reader.ReadToEnd();
  }
}
```

The developer might use `await ReadToEndAsync()` instead of the synchronous `ReadToEnd` call, especially if the stream is expected to be large.

In the example below from iRacerMotionControl [49], the situation is more severe.

```
private async void BT2Arduino_Send(string WhatToSend){
  ...
```

```
    await BTSock.OutputStream.WriteAsync(datab);
    txtBTStatus.Text = "sent";
    System.Threading.Thread.Sleep(5000);  ...
  }
```

The UI event thread calls `BT2Arduino_Send`, which blocks the UI thread by busy-waiting for 5 seconds. Instead of using the blocking `Thread.Sleep` method, the developer should use the non-blocking `Task.Delay(5000)` method call to preserve similar timing behavior, and `await` it to prevent the UI to freeze for 5 seconds.

Open-source corpus. We found 115 instances of potentially long-running operations in 22% of the 167 apps that use `async`. **1 out of 5 apps miss opportunities in at least one async method to increase asynchronicity**.

Proprietary corpus. We found 21 instances of potentially long-running operations in 12 of 545 `async` methods.

4.4.4 Unnecessarily capturing context

`async/await` introduce new risks if the context is captured without specifying `ConfigureAwait(false)`. For example, consider the following example from adsclient [50]:

```
  void GetMessage(byte[] response) {
    ...
    ReceiveAsync(response).Wait(); ...
  }
  async Task<bool> ReceiveAsync(byte[] message) {
    ...
    return await tcs.Task;
  }
```

If `GetMessage` is called from the UI event thread, the thread will wait for completion of `ReceiveAsync` because of the `Wait` call. When the await completes in `ReceiveAsync`, it attempts to execute the remainder of the method within the captured context, which is the UI event thread. However, the UI event thread is already blocked, waiting for the completion of `ReceiveAsync`. Therefore, a deadlock occurs.

To prevent the deadlock, the developer needs to set up the `await` expression to use `ConfigureAwait(false)`. Instead of attempting to resume the `ReceiveAsync` method on the UI event thread, it now resumes on the thread pool, and the blocking wait in `GetMessage` does not cause a deadlock any more. In the example above, although `ConfigureAwait(false)` is a solution, we fixed it by removing `await` because it was also an instance of unnecessary `async/await` use. The developer of the app accepted our fix as a patch.

We found 5 different cases for this type of deadlock which can happen if the caller method executes on UI event thread.

Capturing the context can also cause another problem: it hurts performance. As asynchronous GUI applications 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 thousands of paper cuts. It also enables a small amount of parallelism: some asynchronous code can run in parallel with the UI event thread instead of constantly badgering it with bits of work to do.

To mitigate these problems, developers should await the `Task` with `ConfigureAwait(false)` whenever they can. If the statements after the `await` expression do not update the UI, `ConfigureAwait(false)` must be set. Detecting this misuse is important because using `ConfigureAwait(false)` might prevent future bugs like deadlocks and improve the performance.

Open-source corpus. 1786 out of 2383 `async` methods do not update GUI elements in their call graph after `await` expressions. We found that `ConfigureAwait(false)` is used in only 16 out of these 1786 `async` methods in `await` expressions. All 1770 other `async` methods should have used `ConfigureAwait(false)`. **99% of the time, developers did not use `ConfigureAwait(false)` where this was needed.**

Proprietary corpus. 544 out of 545 `async` methods are library `async` methods so they do not need to capture the context in their bodies. **However, all 544 library `async` methods do not use `ConfigureAwait(false)`.**

Table 4.5: Statistics of `async/await` misuses in the open-source code. The first column shows the total number of idiom instances. The second column contains the total number of apps with idiom instances. The third column shows the relative amount of apps with idiom instances.

Misuse	#	Method	App
(1) Fire & Forget	460	19%	76%
(2) Unnecessary Async	324	14%	26%
(3) Potential LongRunning	115	5%	22%
(4) Unnecessary Context	1770	74%	86%

4.5 Asyncifier: Refactoring from Callbacks to `async/await`

Based on our findings from Section 4.2, developers regularly make mistakes when introducing `async/await` in their Windows Phone apps. To support them, we developed a refactoring tool, Asyncifier that transforms legacy APM-based code into `async`-based code. This gives programmers production-level examples in their own code base of how `async`-based code should look. Additionally, it helps programmers move from the legacy APM pattern to the modern TAP and `async/await` pattern. Section 4.5.1 describes the challenges of refactoring APM-based code to code using `async`. Once those are clear, we describe a canonical form of the original source code to which the refactoring can be applied in Section 4.5.2. Section 4.5.3 describes the refactoring algorithm, as it is applied to code in the canonical form. Finally, Section 4.5.4 describes the most important implementation details of the refactoring tool.

4.5.1 Challenges

There are three main challenges that make it hard to execute the refactoring quick and flawlessly by hand. First, the developer needs to understand if the APM instance is a candidate for refactoring based on the preconditions in Section 4.5.2. Second, he must transform the code while retaining the original behavior of the code - both functionally and in terms of scheduling. This is non-trivial, especially in the presence of (1) exception

handling, and (2) APM `End` methods that are placed deeper in the call graph.

Exception handling. The refactoring from APM to `async/await` should retain the functional behavior of the original program, both in the normal case and under exceptional circumstances. In 52% of all APM instances, `try-catch` blocks are in place to handle those exceptions. The `try-catch` blocks surround the `End` method invocation, which throws an exception if the background operation results in an exceptional circumstance. These `catch` blocks can contain business logic: e.g., a network error sometimes needs to be reported to the user: *"Please check the data or WiFi connection"*. Code 4.1 shows such an example.

Code 4.1 EndGetResponse in try-catch block

```
1  void Button_Click(...) {
2    WebRequest request = WebRequest.Create(url);
3    request.BeginGetResponse(Callback, request);
4  }
5  void Callback(IAsyncResult ar) {
6    WebRequest request = (WebRequest)ar.AsyncState;
7    try {
8      var response = request.EndGetResponse(ar);
9      // Code does something with successful response
10   } catch (WebException e) {
11     // Error handling code
12   }
13 }
```

Code 4.2 EndGetResponse on longer call graph path

```
1  void Button_Click(...) {
2    WebRequest request = WebRequest.Create(url);
3    request.BeginGetResponse(ar => {
4      IntermediateMethod(ar, request);
5    }, null);
6  }
7  void IntermediateMethod(IAsyncResult result, WebRequest request) {
8    var response = GetResponse(request, result);
9    // Code does something with response
10 }
11 WebResponse GetResponse(WebRequest request, IAsyncResult result) {
12   return request.EndGetResponse(result);
13 }
```

The naive approach to introducing `async/await` is to replace the `Begin` method invocation with an invocation to the corresponding TAP method, and await the result immediately. However, the `await` expression is the site that can throw the exception when the background operation failed. In the APM code, the exception would be thrown at the `End` call site. Thus, the exception would be thrown at a different site, and this can drastically change behavior. Exception handling behavior can be retained by introducing the `await` expression as replacement of the `End` method call at the exact same place. This is a non-trivial insight for developers, because online examples of `async/await` only show the refactoring for extremely simple cases, where this is not a concern.

Hidden (Nested) `End` method calls. The developer needs to take even more care when the `End` method is not immediately called in the callback lambda expression, but is 'hidden' deeper down the call chain. In that case, the `Task` instance that is returned by the TAP method that replaces the `Begin` method, must be passed down to where the `End` method invocation was, to retain exceptional behavior. This requires an inter-procedural analysis of the code: each of the methods, through which the `IAsyncResult` 'flows', must be refactored, which makes the refactoring more tedious. The developer must trace the call graph of the callback to find the `End` method call, and in each encountered method: (1) replace the `IAsyncResult` parameter with a `Task<T>` parameter (with `T` being the return type of the TAP method, (2) replace the return type `R` with `async Task<R>`, or `void` with `async void` or `async Task`, and (3) stay on the same thread by ignoring the current synchronization context through the introduction of `ConfigureAwait(false)` at each await expression.

As shown in the results of the empirical study, when its presence is important to retain UI responsiveness, developers almost never use `ConfigureAwait(false)` where it should be used. Code 4.2 shows an example of a nested `End` method call.

4.5.2 Canonical form of APM-based code

A key insight in designing refactorings is that there is a canonical form for input code. This canonical form adheres to the preconditions of the refactoring transformation, so that the

result of the transformation is indeed correct. However, many APM instances are not in the canonical form, but are very close to the canonical form. This means that, often, other refactorings are necessary to transform the input code into canonical form. First, we describe the preconditions of the algorithm of our refactoring transformation. Then, we show a few examples of such preparatory refactorings.

Algorithm preconditions. The refactoring has several preconditions, which together dictate the canonical form which the source code must have for a successful transformation. It starts with an invocation of an APM `Begin` method. It is a candidate for refactoring to `async`, if it adheres to the following preconditions and restrictions:

P1: *The APM method call must represent an asynchronous operation for which a TAP-based method also exists.*

Obviously, if the corresponding TAP-based method does not exist, the code cannot be refactored.

P2: *The `Begin` method invocation statement must be contained in a regular method, i.e, not in a lambda expression or delegate anonymous method.*

The method containing the `Begin` invocation will be made `async`. While it is possible to make lambdas and delegate anonymous methods `async`, this is considered a bad practice because it usually creates an `async void` fire & forget method (see Section 4.4.1).

P3: *The callback argument must be a lambda expression with a body consisting of a block of statements.*

The call graph of that block must contain an `End` method invocation that takes the lambda `IAsyncResult` parameter as argument. This means that the callback must actually end the background operation.

P4: *In the callback call graph, the `IAsyncResult` lambda parameter should not be used, except as argument to the `End` method.*

The `IAsyncResult` lambda parameter will be removed after the refactoring.

Code 4.3 An APM instance that adheres to preconditions

```
void Action(WebRequest request) {
  request.BeginGetResponse(asyncResult => {
    var response=request.EndGetRequest(asyncResult);
    // Code does something with response
  }, null);
}
```

Code 4.4 Refactored version of Code 4.3

```
void Action(WebRequest request) {
  var task = request.GetResponseAsync();
  var response = await task.ConfigureAwait(false);
  // Code does something with response
}
```

P5: *The state argument must be a null literal.*

As the `IAsyncResult` lambda parameter must be unused, its `AsyncState` property should be unused as well, so the state argument expression of the `Begin` method invocation should be null.

P6: *The* `IAsyncResult` *return value of the* `Begin` *method invocation should not be used in the initiating method (the method containing the* `Begin` *method invocation).*

As the `Begin` method invocation disappears, its return value should not be used.

Code 4.3 shows a valid example in the context of these preconditions. It can be refactored with our tool to take advantage of async/await, as shown in Code 4.4.

Refactoring to the canonical form. In real-world applications, we found that many instances are not in the canonical form as described in Section 4.5.2. The preconditions listed in that section would restrict applicability of the refactoring significantly. Fortunately, many instances in other forms can be refactored into the canonical form with well-known refactorings.

Code 2.7 shows an example that fails **P3** and **P5**: the callback argument is a method reference, and the state argument is not null. This instance can be refactored into the code shown in Code 4.5 by applying the *Introduce Parameter* refactoring to remove the `request` variable and add it as parameter to the original `Callback` method.

Based on encountered cases in the analyzed code corpus, we have identified and (partially) implemented several such refactorings. Examples are (1) identification of unused state argu-

Code 4.5 Code 2.7 after applying *Introduce Parameter*

```
void GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  request.BeginGetResponse(asyncResult => {
    Callback(asyncResult, request);
  }, null);
}
void Callback(IAsyncResult ar, WebRequest request) {
  var response = request.EndGetResponse(ar);
  var stream = response.getResponseStream();
  var content = stream.ReadAsString();
  Dispatcher.BeginInvoke(() => {
      textBox.Text = content;
  });
}
```

ments which can be replaced with `null` (solves violations of **P5**), and (2) rewriting of some callback argument expressions (solves violations of **P3**).

4.5.3 Refactoring algorithm

Asyncifier detects all `Begin` method invocations that fulfill the preconditions. It takes the following steps to refactor the APM instance to `async`-based code:

Traveling the call graph from APM begin to end. First, Asyncifier explores the call graph of the body of the callback lambda expression to find the invocation path to the `End` invocation. It does a depth-first search of the call graph, by looking up the symbols of any non-virtual method that is encountered. There are two possible scenarios: the `End` method invocation (1) is placed directly in the lambda expression, or (2) it is found on the call graph of the lambda body in another method's body. Code 4.3 is an example of the first case.

In the second case, Asyncifier identifies three different methods which are on the call graph path: (1) the *initiating method*, i.e., the method containing the `Begin` method invocation, (2) the *result-obtaining method*, i.e., the method containing the `End` method invocation, and (3) *intermediate methods*, i.e., the remaining methods on the path. Code 4.5 is an example of the second case. We use this example in the description of the following steps.

Rewriting the initiating method. In both cases, the *initiating method* needs to be rewritten. The tool adds the `async` modifier to the signature of the initiating method. It

changes the return value is to either *Task* instead of *void*, or *Task*<*T*> for any return type *T*.

```
void GetFromUrl(string url) { ... }
```

```
async Task GetFromUrl(string url) { ... }
```

Then, the `Begin` method invocation must be replaced with the corresponding TAP method invocation. The `Begin` invocation statement is replaced with a local variable declaration that captures the `Task` return value of the TAP method invocation. The (optional) `Task` parameter type is the return type of the `End` method:

```
request.BeginGetResponse(...);
```

```
Task<WebResponse> task = request.GetResponseAsync();
```

It then concatenates the statements in the lambda expression body to the body of the initiating method:

```
async Task GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  var task = request.GetResponseAsync();
  Callback(asyncResult, request);
}
```

Finally, it replaces the lambda parameter reference `asyncResult` with a reference to the newly declared `Task` instance. The rewritten initiating method:

```
async Task GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  var task = request.GetResponseAsync();
  Callback(task, request);
}
```

Rewriting the result-obtaining method. If the APM `End` method invocation was found in a result-obtaining method, the refactoring updates the signature of that method as follows: (1) it adds the `async` modifier, (2) it replaces return type `void` with `Task`, or any other `T` with `Task<T>`, and (3) it replaces the `IAsyncResult` parameter with `Task<R>`, with `R` the return type of the `End` method.

```
void Callback(IAsyncResult asyncResult,
              WebRequest request) { ... }
```

```
async Task Callback(Task<WebResponse> task,
                    WebRequest request) { ... }
```

Then it replaces the `End` method invocation expression with `await task`, without capturing the synchronization context:

```
var response = request.EndGetResponse(asyncResult);
```

```
var response = await task.ConfigureAwait(false);
```

The refactoring rewrites the complete APM instance into the code shown in Code 4.6. If the introduction of new variables leads to identifier name clashes, ASYNCIFIER disambiguates the newly introduced names by appending an increasing number to them, i.e., *task1*, *task2*, etc.

Callbacks containing the `End` call. If the `End` method invocation is now in the initiating method, ASYNCIFIER replaces it with an await expression, and the refactoring is complete. The example in Code 4.3 would be completely refactored at this point:

If there is no result-obtaining method, but instead the `End` method invocation is directly in the callback of the `Begin` method, the previous step is skipped. The `End` method invocation was inlined into the initiating method in step 2 (see Section 4.5.3). The refactoring replaces the `End` method invocation with an `await` expression, and the refactoring is complete. At

this point, the example in Code 4.3 would be completely refactored. Code 4.4 shows the refactored code.

Rewriting intermediate methods. Intermediate methods must be rewritten if the `End` method is not invoked in the callback lambda expression body. Every method on the call graph path from the initiating method to the result-obtaining method is recursively refactored. The same steps are applied as for the result-obtaining method. Additionally, at the call site of each method, the reference to the (removed) `result` parameter is replaced with a reference to the (newly introduced) `task` parameter.

Retaining original behavior. First, it is crucial that the refactored code has the same behavior in terms of scheduling as the original code. By calling the `Begin` method or the TAP method, the asynchronous operation is started. In the APM case, the callback is only executed once the background operation is completed. With `async`, the same *happens-before* relationship exists between the `await` expression and the statements that follow the `await` of the `Task` returned by the TAP method. Because the statements in callbacks are placed after the `await` expression that pauses execution until the completion of the background operation, this timing behavior is preserved.

Second, after the refactoring, the different code pieces will be executed on the same thread as they were executed before the refactoring. The `ConfigureAwait(false)` flag is used to make sure this is the case. This makes sure that the asynchronous behavior of the application, and thus its responsiveness stays the same.

Finally, because the await expression is positioned at the same place as the APM `End` method invocation, behavior in the presence of thrown exceptions remains the same.

Note on call sites of the initiating method. When the original initiating method has a return type `T`, in call sites of that method, the return value might be captured. After executing the refactoring transformation, the return type of the initiating method will be `Task<T>`. These call sites will need to be manually adjusted (the IDE or compiler will emit a type error on compilation). Such call sites can simply use `Task.Result`, or the containing method could also be made into an asynchronous method.

Code 4.6 async/await code after refactoring Code 2.7

```
async Task GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  Task<WebResponse> task=request.GetResponseAsync();
  Callback(task, request);
}

async Task Callback(Task<WebResponse> task,
                    WebRequest request) {
  var response = await task.ConfigureAwait(false);
  var stream = response.getResponseStream();
  var content = stream.ReadAsString();
  Dispatcher.BeginInvoke(() => {
      textBox.Text = content;
  });
}
```

4.5.4 Refactoring implementation

We have implemented the APM to `async/await` refactoring in C# with Microsoft's Roslyn framework. It can be downloaded from the Visual Studio Gallery (the official store for extensions for VS) [1].

Implementation limitations. The current tool is a research prototype aimed at showing the feasibility of the refactoring for common cases. Because of that, the current implementation is based on the following assumptions related to re-use of `Begin` or `End` methods. First, there should not be other call graph paths leading from any `Begin` method call to the target `End` method, which means so much as that the specific `End` method invocation must not be shared between multiple `Begin` invocations. Second, recursion in the callback through another `Begin` call that references the same callback again is not allowed (essentially, this is also sharing of an `End` method call). Third, the refactoring does not support multiple `End` method invocations that correspond to a single `Begin` method invocation, for example through the use of branching. However, this case is very rare. With additional engineering effort support for these cases can be created as well, which is outside the scope of this thesis.

[1] https://marketplace.visualstudio.com/items?itemName=SemihOkur.Asyncifier

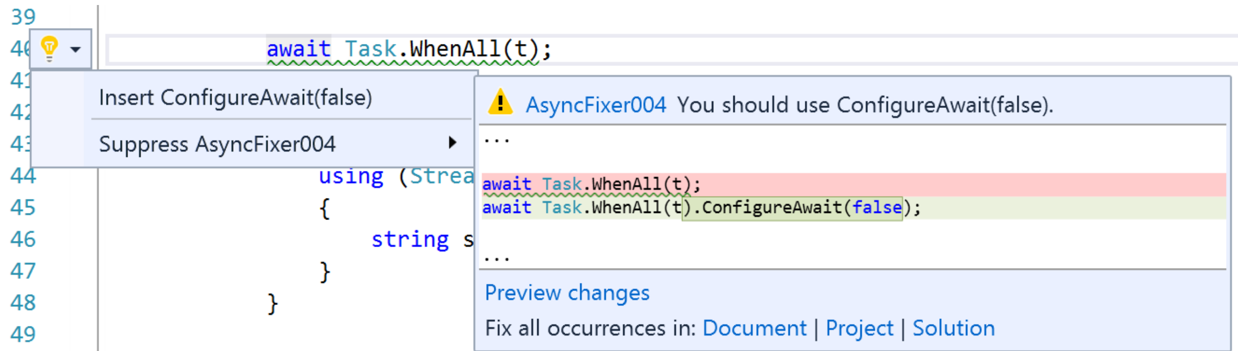

Figure 4.1: AsyncFixer tool interactivity

4.6 AsyncFixer: Fixing Common Misuses

Asyncifier helps developer to upgrade APM code to `async/await` without errors by retaining the original behavior. If the developer manually introduces or has already introduced `async`, we implemented another tool that detects and corrects common misuses: AsyncFixer.

AsyncFixer will help in finding and removing misuses that we explained in Section 4.4. We implemented AsyncFixer by using Roslyn Diagnostics API. There are two options to use AsyncFixer:

VSIX extension. In this option, AsyncFixer will work just in the Visual Studio and work as an analyzer on every project developers open in Visual Studio. This mode is also called as live code analysis because AsyncFixer is constantly scanning the source code as soon as the user typed in new code. Thus, it dramatically shortens the time between the introduction of an error and its detection.

When AsyncFixer detects a misuse, the light bulb will appear in the left margin and developers can click on it or hover over the code to see suggestions and code fixes as shown in Figure 4.1. AsyncFixer can also operate in batch mode to correct all misuses in the document, project, or solution file as shown in Figure 4.1. AsyncFixer VSIX extension can be downloaded from the Visual Studio Gallery [2].

[2] `https://marketplace.visualstudio.com/items?itemName=SemihOkur.AsyncFixer`

NuGet package. AsyncFixer can also be deployed as a package (library) for the NuGet package management system [34]. In this option, AsyncFixer will work as a project-local analyzer that participates in builds. Attaching an analyzer to a project means that the analyzer travels with the project to source control and so it is easy to apply the same rule for the team. It also means that commandline builds report the issues reported by the analyzer. The NuGet package can be downloaded from the NuGet Gallery [3].

4.6.1 Fire & forget methods

There is no automated fix for this misuse. AsyncFixer just converts the return type from `void` to `Task`. However, the fire & forget `async` method should also be awaited in all callers of it, causing changing the semantics. Therefore, the developer's understanding of code is required to fix this case.

4.6.2 Unnecessary `async/await` methods

AsyncFixer checks whether `async` method body has only one `await` keyword and this `await` is used for a TAP method call that is the last statement of the method. AsyncFixer does not do this for `async void` (fire & forget) methods; because if it removes `await` from the last statement in `async void` methods, it will silence the exception that can occur in that statement.

To fix these cases, AsyncFixer removes the `async` from the method identifiers and the `await` keyword from the TAP method call. The method will return the `Task` that is the result of TAP method call.

4.6.3 Using long-running operations under `async` methods

To detect these operations, AsyncFixer looks up symbols of each method invocation in the bodies of `async` methods. After getting symbol information, AsyncFixer looks at the other members of the containing class of that symbol to check whether there is an asynchronous

[3] `https://nuget.org/packages/AsyncFixer`

version. For instance, if there is an `x.Read()` method invocation and x is an instance of the `Stream` class, AsyncFixer looks at the members of the `Stream` class to see whether there is a `ReadAsync` method that gets the same parameters and returns Task. By checking the members, AsyncFixer can also find asynchronous versions not only in the .NET framework but also in third-party libraries.

AsyncFixer also maps corresponding blocking and non-blocking methods which do not follow the `Async` suffix convention (e.g., `Thread.Sleep` ->`Task.Delay`).

AsyncFixer avoids introducing asynchronous operations of file IO operations in loops, as this could result in slower performance than the synchronous version.

After finding the corresponding non-blocking operation, Asyncifier simply replaces the invocation with the new operation and makes it `await`'ed.

4.6.4 Unnecessarily capturing context

AsyncFixer checks the call graph of `async` methods to see if there are accesses of GUI elements. All GUI elements are in the namespaces `System.Windows` and `Microsoft.Phone`. If AsyncFixer does not find any use of elements from those namespaces after `await` expressions, it introduces `ConfigureAwait(false)` in those await expressions.

4.7 Evaluation

To evaluate Asyncifier and AsyncFixer, we studied their *applicability*, their *impact* on the code, their *performance*, and the *usefulness* of their results to developers.

4.7.1 Applicability

We executed Asyncifier in batch mode over the entire open-source code corpus that we described in Section 4.2.1. The code corpus contains 1245 APM instances, i.e., invocations of an APM `Begin` method. Of these 1245 instances, 54% adhered to the preconditions defined in Section 4.5.2. Only the APM instances that passed the preconditions test were

refactored, and then compiled in-memory to verify that the AST transformation created a legal C# program. For the other 46% of APM instances, the original code was untouched. We verified the success of the refactorings by manually checking the output of the batch tool on 10% of the refactored instances (randomly sampled).

There are two main reasons why the refactoring tool could not successfully refactor 46% of the APM instances. First, some instances do not adhere to the preconditions set in Section 4.5.2. Second, the tool currently has limitations, as described in Section 4.5.4. The former are mostly instances that cannot be refactored because of fundamental limitations of the algorithm. One example are callback expressions that reference a (public) field delegate that is perhaps set by external code. Another example are APM `End` methods that are hidden behind interface implementations (both violations of precondition **P3**).

We also applied AsyncFixer to the open-source code corpus (2209 times). All instances of type 2, 3, and 4 misuses (see Table 4.5 were corrected automatically.

4.7.2 Impact on source code

To evaluate the impact of our refactorings on the code we investigate the size of the changes. Asyncifier changes 28.9 lines on average per refactoring. This shows that automation is needed, since each refactoring changes many lines of code. Moreover, these changes are not trivial.

AsyncFixer changes a single line of source code, in the case of use of a synchronous method call in an asynchronous methods (see Section 4.6.3) or a missing `ConfigureAwait(false)` setting (see Section 4.6.4). The unnecessary use of `async/await` (see Section 4.6.2) required changing two or three lines of source code. However, the main advantage of an in-IDE implementation is not that the change to be made is hard, but primarily that misuse becomes visible to the developer.

4.7.3 Tool performance

For Asyncifier, the average time needed to refactor one instance is 508ms. This is fast enough to make the refactoring suitable for implementation as an interactive refactoring in an IDE. Because the number of APM instances per application is almost never higher than 100, Asyncifier is fast enough for batch-mode refactoring too.

The quick fixes for unnecessary use of `async`, and for detection of synchronous method calls in asynchronous code are straightforward, so we did not measure computation time for those. However, it is complicated to decide whether a missing `ConfigureAwait(false)` (see Section 4.6.4) setting is a misuse, because the call graph of the asynchronous method needs to be inspected. Still, it only took on average 47 ms to analyze a single `await` expression. This is fast enough to interactively use the analysis in an IDE.

4.7.4 What do developers think?

To further evaluate the usefulness in practice, we conducted a qualitative analysis of the 10 most recently updated apps that have APM instances. We applied Asyncifier ourselves, and offered the modifications to the original developers as a patch via a pull request.[4] Of 9 out of 10 apps, the developers responded, and they accepted each of the pull requests, for in total 28 refactored APM instances.

We received very positive feedback from the developers on those pull requests. One developer *"look[s] forward to the release of that refactoring tool, it seems to be really useful."*[51]. The developer of phoneguitartab [52] 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 the refactoring from APM to `async`.

For AsyncFixer, we selected the 10 most recently updated apps containing either unnecessary uses of `async/await` or use of synchronous method calls in asynchronous methods. This led to a list of 19 apps (one app was on the list for both misuses). We did not separately select apps for missing `ConfigureAwait(false)` instances. For each of these apps, we ran AsyncFixer in batch mode and supplied the changes to the developers as patch or pull

[4] All patches can be found on our web site: `http://LearnAsync.NET`

Table 4.6: AsyncFixer patches accepted by developers

Section	Misuse	#
4.6.2	Unnecessary Async	149
4.6.3	Potential LongRunning	39
4.6.4	Unnecessary Context	98
	Total	286

request. We received a response from developers of 18 of the 19 apps, who accepted all 286 proposed changes. Table 4.6 shows a breakdown of the types of accepted changes.

The responses from developers to the proposed changes were similarly positive. One developer pointed out that he missed several unnecessary `async/await` instances that AsyncFixer detected: *"Totally agree, I normally try to take the same minimizing approach, though it seems I missed these."* [53]. The developer of SoftbuildData [54] experienced performance improvements after removing unnecessary `async`: "[...] performance has been improved to 28 milliseconds from 49 milliseconds." Again, these illustrate the need for tools that support the developer in finding problems in the use of `async`.

Some other positive responses from the developers are the following: *"Thanks for your great advice"* [55], *"This is awesome insight! it's working like wonders."* [48], and *"That's very interesting and useful."* [56].

Furthermore, the developers of the playerframework [57] said that they missed the misuses because the particular code was ported from old asynchronous idioms. This demonstrates the need for Asyncifier as it can help a developer to upgrade his or her code, without introducing incorrect usage of `async`.

4.8 Discussion

Our study has implications for developers, educators, language and library designers, tool providers, and researchers.

Developers learn and *educators* teach new programming constructs through both positive

and negative examples. Robillard and DeLine [58] study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples. We provide hundreds of real-world examples of all asynchronous idioms on `http://LearnAsync.NET`. Because developers might need to inspect the whole source file or project to understand the example, we also link to highlighted source files on GitHub. We also provide negative examples anonymously, without giving app names.

Language and library designers can learn which constructs and idioms are embraced by developers, and which ones are tedious to use or error-prone. Because some other major languages (Java 9) have plans to introduce similar constructs for asynchronous programming, this first study can guide them to an improved design of similar language constructs for their languages. The architects of async constructs in C#, F#, and Scala confirmed that our findings are useful and will influence the future evolution of these constructs. For instance, capturing the context need not be the default: as we have seen developers are very likely to forget to use `ConfigureAwait(false)`.

Tool providers can take advantage of our findings on `async/await` misuse. IDEs such as Visual Studio should have built-in quick fixes (similar to ours) to prevent users from introducing misuse. For instance, if developers introduce a fire & forget method, the IDE should give a warning unless the method is the top level event handler.

Researchers in the refactoring community can use our findings to target future research. For example, as we see from Table 4.2, the usage of `Task` jumped to 8% from 1% in WP8. This calls for work on a tool that converts old asynchronous idioms of CPU-bound computations (e.g., `Thread`) to new idioms (e.g., `Task`).

4.8.1 Why is `async/await` commonly misused?

We have seen extensive misuse of the `async/await` keywords in the projects in our code corpus. Different authors, both from Microsoft and others, have documented these potential misuses extensively. This raises the question: Why is the misuse so extensive? Are developers just uninformed, or are they unaware of risks or performance characteristics of `async`?

The `async/await` feature is a powerful abstraction. Asynchronous methods are more

complicated than regular methods in three ways.

1. Control flow of asynchronous methods: Control is returned to the caller when awaiting, and the continuation is resumed later on.

2. Exception handling: Exceptions thrown in asynchronous methods are automatically captured and returned through the `Task`. The exception is then re-thrown when the `Task` is awaited.

3. Non-trivial concurrent behavior: Up to the first await expression, the asynchronous method is executed synchronously. The continuation is potentially executed in parallel with the main thread. Each of these is a leak in the abstraction, which requires an understanding of the underlying technology - which developers do not yet seem to grasp.

Another problem might simply be the naming of the feature: asynchronous methods. However, the first part of the method executes *synchronously*, and possibly the continuations do so as well. Therefore, the name *asynchronous* method might be misleading: the term *pauseable* could be more appropriate.

Microsoft has introduced a powerful new feature with `async/await` keywords in C# 5. This empowers developers. However, the flip side of the same coin is that they can easily misuse it. Therefore, it is important to support developers in their understanding of the feature, its pros and cons, and best practices. Documentation is one manner of doing this; a toolkit and/or IDE support is another. We believe that these are complimentary, and should both be provided.

4.8.2 Threats to validity

External. For what C# programs are the results representative? We analyzed both industry and open-source code bases. Open-source 1378 Windows Phone apps span a wide domain, from games, social networking, and office productivity to image processing and third party libraries. They are developed by different teams with 3376 contributors from a large

and varied community, comprising all Windows Phone apps from GitHub and Codeplex. Our industry code base consists of reusable libraries, server side code (ASP.NET), and console apps.

Our *tools* can also be applied to any other C# system such as tablet and desktop apps.

Internal. Are the measurement results that we obtained actually informing us on what we want to know? Or are they skewed in some way? We wanted to know how programmers use and misuse asynchronous programming constructs. We studied construct use through static analysis: we counted the number of call site instances. The results do not give insight into how *hot* those call sites are, or how much CPU time is spent in asynchronous parts of the program. In that case, using dynamic analysis would have been more appropriate. However, we were not interested in the performance perspective. Instead, we were interested in the development and maintenance point of view: development, comprehension and evolution of code.

With respect to the misuse analysis, for most cases, our analysis is an under-estimation: For example, we detect one form of *unnecessary* `async`, but likely there are other unnecessary uses that we do not detect. In other cases, heuristics are used, for example to discover the existence of asynchronous versions of long running methods, or for detecting GUI code. So we can conclude that the results are a lower bound on misuse numbers, and are likely to be slightly higher.

Concerning the applicability of the refactorings, our tool presently can handle 54% of the APM cases. This percentage can go up by addressing some of Asyncifier's tool limitations.

Reliability. To facilitate replication of our analysis, we provide a detailed description of our results with fine-grained reports online at `http://LearnAsync.NET`. It shows all usage examples that we found of asynchronous idioms (e.g., APM, TAP, `Thread`, `ThreadPool`, etc.). It also shows misuse examples of `async`.

Asyncifier and AsyncFixer can be downloaded through our webpage or Visual Studio Gallery.

4.9 Summary

Because responsiveness is getting very important, asynchronous programming is already a first-class citizen in modern programming languages. However, the empirical research community and tool providers have not yet caught up.

Our large-scale empirical study of C# apps in both industry and open-source codebases provides insight into how developers use asynchronous programming. We have discovered that developers make many mistakes when manually introducing asynchronous programming. We provide a toolkit to support developers in preventing and curing these mistakes. Our toolkit (1) safely refactors legacy callback-based asynchronous code to the new `async`, (2) detects and fixes existing errors, and (3) prevents introduction of new errors. Evaluation of the toolkit shows that it is highly applicable, and developers already find the transformations very useful and are looking forward to using our toolkit. We hope that our study motivates other follow-up studies to fully understand the state of the art in asynchronous programming.

CHAPTER 5

Finding and Correcting Misused Asynchrony

5.1 Introduction

Asynchronous programming is crucially important for today's programs. Programs that are UI-centric such as many desktop and mobile apps use asynchrony to improve responsiveness. Programs that need to service many requests (e.g., web, console, or libraries) use asynchrony for scalability. Thus, many developers now treat asynchrony as a first-class citizen in their programs.

All major software development platforms support asynchronous programming through APIs that rely on callbacks. While these APIs make asynchrony possible, they do not make it easy. Callbacks invert the control flow and introduce accidental complexity [9]. To simplify asynchronous programming, some languages (C# [10], F# [9], Scala [11]) introduced `async/await` keywords. These keywords are novel and many developers are not familiar with them.

In 2013, we were the first to study [15] the novel `async/await` in C#. Back then, the main problem was *under-use*: programs used it sparingly, and it was not popular outside the domain of mobile apps. This inspired us to develop a refactoring tool [15] that helps developers embrace it. In our current formative study comprising 51M SLOC representing all domains of computing, we found a 10-fold increase in usage of `async/await` since 2013. By 2016, developers across all domains embraced `async/await`. Are developers now using `async/await` correctly?

In this paper we first conduct a large scale formative study to understand how developers use `async/await` in both open-source and proprietary programs across all domains. Our formative study answers two research questions:

RQ1: *How are developers embracing modern asynchronous constructs?* We analyzed annual code snapshots of the 100 most popular C# repositories in Github [16] from 2013 to 2016, comprising 51M SLOC in total. We found that developers fully embraced modern, `async/await`-based constructs, surpassing the usage of legacy, callback-based constructs.

RQ2: *To what extent do developers misuse modern asynchronous constructs?* We discovered 10 kinds of prevalent misuses that have not been studied before.

These `async/await` misuses have severe consequences. Some cause subtle data-races that are unique to `async/await` and cannot be detected by previous race detecting techniques. Other misuses "swallow" the run-time uncaught exceptions thrown by a program which hinder developers when searching for errors. Other misuses significantly degrade the performance of async programs or can even deadlock the program. Other misuses make developers erroneously think that their code runs sequentially, when in reality the code runs concurrently; this causes unintended behaviour and wrong results.

The `async/await` misuses are a new source of bugs in software that the research community has not studied before. Moreover, most of the misuses that we discovered cannot be detected by the existing bug-finding tools. This paper is the first to raise the awareness of the research community on an increasingly important source of bugs. We hope that it serves as a call to action for the testing, program verification, and program analysis community.

Inspired by our formative study, we extended AsyncFixer, our existing static analysis tool, (i) *to detect* 10 more kinds of `async/await` misuses, and (ii) *to fix* them via program transformations. Developers can use AsyncFixer as *live* code analysis: it constantly scans the source code as soon as the user types in new code. A key challenge is to make the detection efficient so that developers can use AsyncFixer interactively. The static analysis that detects the 10 kinds of misuses is non-trivial: it requires control- and data-flow, as well as dependence information. A syntactic find-and-replace tool cannot detect these misuses.

To empirically evaluate AsyncFixer, we ran it over a diverse corpus of 500 programs from GitHub, comprising 24M SLOC. We also ran it over the code-bases of two industrial partners, comprising 4M SLOC. AsyncFixer detected 891 previously unknown misuses in open-

source software, and 169 in proprietary software. The open-source developers accepted 41 AsyncFixer-generated patches. Moreover, our industrial partners integrated AsyncFixer directly into their automated build process, so that misuses are caught and fixed before they are in production code.

5.2 Formative Study

We first conducted a formative study to understand the needs of programmers with regard to asynchronous programming. To conduct our study, we created a code corpus from open-source programs.

Methodology. We chose Github [16] as the source of the corpus because Github is the most popular open-source hosting service. We downloaded the git repositories of the most popular (i.e., sorted by stars) 100 C# programs that were created before January 1, 2013 and have at least one commit since January 1, 2016. The code corpus has various programs which target library, console, windows phone, windows store, and ASP.NET (web) platforms.

5.2.1 **RQ1:** How are developers embracing modern asynchronous constructs?

We wonder how soon developers started to use modern async constructs and whether the usage of modern constructs has surpassed the usage of legacy constructs. We developed a static analysis tool to get the usage statistics of async constructs. The tool analyzed annual code snapshots of our corpus starting from 2013 to 2016 because modern async constructs were released in 2012. In total, the tool analyzed 4 different versions of 100 programs, which comprise 51M SLOC.

Async I/O constructs. .NET provides three kinds of constructs to asynchronously execute I/O operations (see Section 2.2.2 for more details). The tool measured the call-sites of APM, EAP, and TAP methods. As shown in Figure 5.1, the usage of legacy async I/O constructs remained constant, but the usage of TAP grew a lot. It means that developers

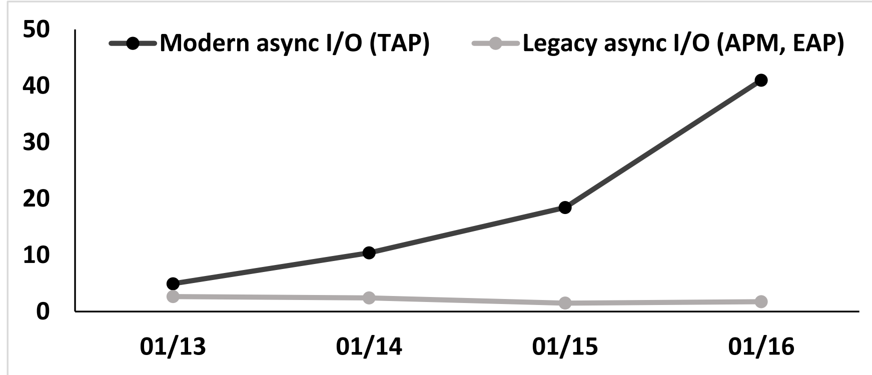

Figure 5.1: Usage trend of `async` I/O constructs over four years. Y-axis represents the number of call-sites per 100K lines of code.

Table 5.1: Usage trend of `async/await` over four years. The second row represents the number of `async` methods per 100K lines of code.

01/2013	01/2014	01/2015	01/2016
4.9	13.4	23.4	50

are not replacing legacy constructs with modern ones but adding even more asynchrony in the form of TAP.

Async CPU-bound constructs. .NET is a rich platform for developers to offload custom operations to another thread. We have measured the call-sites of several constructs that represent five approaches explained in Section 2.2.1. For instance, for running a task, we have measured the call-sites of `Task.Run`, `TaskFactory.StartNew`, `Task.Start` methods.

Figure 5.2 shows that the usage of `Thread` and `ThreadPool` decreased, whereas the usage of modern `Task` constructs constantly increased.

Language constructs. TAP constructs from the first group and `Task` constructs from the second group can be used with `async/await` keywords. We measured the number of method definitions that have `async` modifier. As shown in Table 5.1, there is a 10-fold increase in usage of `async` methods since 2013.

The usage of legacy constructs remained the same or decreased. Developers fully embraced modern async constructs, surpassing the usage of legacy constructs.

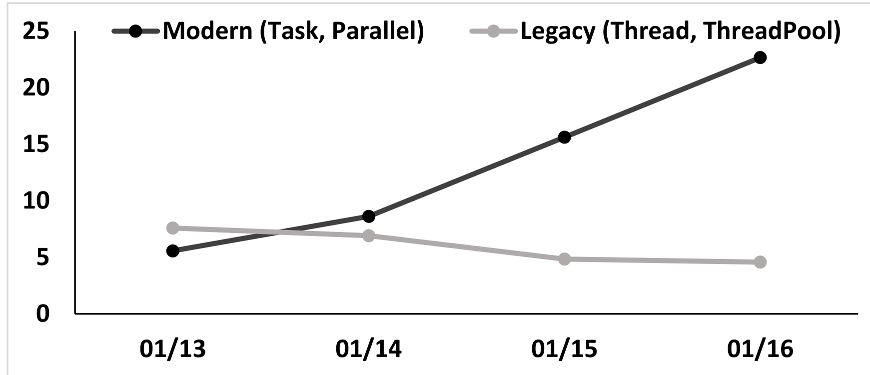

Figure 5.2: Usage trend of `async` CPU-bound constructs over four years. Y-axis represents the number of call-sites per 100K lines of code.

5.2.2 **RQ2:** To what extent do developers misuse modern asynchronous constructs?

In our previous study [15] conducted in 2013, we presented a refactoring tool that migrate the legacy code to `async/await`. In that study, we found little use of `async/await` in programs beyond mobile; and even there we found many instances of anti-patterns (lightweight, stylistic problems such as unnecessary usage of `async/await`) in mobile apps. By 2016, developers across all domains embraced `async/await` and the usage of `async/await` is increased 10-fold. We now expect that we will find prevalent, significant misuses of `async/await` across all domains.

We collected the misuses in the following ways:

1. We manually examined the bug reports of programs in our corpus. We filtered the reports whose content contains `async/await` keywords.

2. We visited two large software companies and talked to some of their developers about the difficulties and most common bugs of modern async constructs.

3. We discovered some misuses ourselves.

We discovered 10 kinds of prevalent misuses that have not been studied before and categorized them into three main types: performance (Section 5.4), correctness (Section 5.5), and bad-practices (Section 5.6).

5.3 AsyncFixer: Extending it to Fix 10 New Misuses

In our previous study (see Chapter 4), we developed AsyncFixer to detect and fix four kinds of `async/await` misuses. We now extended AsyncFixer to detect and fix 10 more kinds of `async/await` misuses.

AsyncFixer is shipped in two different ways: Visual Studio extension (VSIX) and NuGet package.

Nuget package. AsyncFixer can be deployed as a package (library) for the NuGet package management system [34]. In this option, developers add the AsyncFixer nuget package as a dependency to the projects that they want to analyze. Then, AsyncFixer works as part of the automated build process. Build output shows the misuses detected by AsyncFixer.

VSIX package. The VSIX package only works in Visual Studio so command-line builds do not report the misuses unlike the Nuget package. The advantage is that AsyncFixer is enabled for every project that developers open in Visual Studio.

Both packages. Either mode can work as live code analysis in the Visual Studio because AsyncFixer constantly scans the source code as soon as the user types in new code. Thus, it dramatically shortens the time between the introduction of an error and its detection.

When AsyncFixer detects a misuse, the light bulb will appear in the left margin and developers can click on it or hover over the code to see suggestions and code fixes. AsyncFixer can also operate in batch mode to correct all misuses in the document, project, or solution file.

Misuse catalog. There are 10 new kinds of misuses that AsyncFixer can automatically detect and fix. In the next sections, we explain these 10 misuses in detail. For each misuse, we first explain (i) the problem and when the misuse happens, (ii) the static analysis algorithm to detect this misuse, and (iii) the program transformation algorithm to fix it. (see Section 2.3 for the definitions of the terms which are frequently used when explaining the misuses.)

5.4 Performance Misuses

There are misuses which have negative impacts on the performance of the software system. In other words, code could be written differently to improve performance.

5.4.1 Missing parallelism when chaining multiple I/O operations

Problem. Async I/O is one critical way to make the code more scalable. However, developers might still miss opportunities even though they use *async I/O calls*.

In Code 5.1, two independent *async I/O calls* are executed one after another: `GetStorageFolder()` and `GetFiles()`. However, both operations can be run asynchronously like in Code 5.2.

More `await` statements also mean more complexity in the state machine that is generated by the C# compiler, causing more overhead. Hence, developers can gain performance here by (i) lowering the overhead by decreasing the `await` statements and (ii) executing *async I/O calls* in parallel.

Code 5.1 Real-world example for 5.4.1 from LanguageDetectApp [60]

```
1 override async void OnNavigatedTo(NavigationEventArgs e)
2 {
3     ...
4     await _fileViewModel.GetStorageFolder();
5     await _fileViewModel.GetFiles();
6     ...
7 }
```

Code 5.2 Fixed version of Code 5.1

```
1 override async void OnNavigatedTo(NavigationEventArgs e)
2 {
3     ...
4     await Task.WhenAll(_fileViewModel.GetStorageFolder(),
5         _fileViewModel.GetFiles());
6     ...
7 }
```

Detection. AsyncFixer analyzes each method call in the *async methods*. If it detects 2 or more consecutive *awaited async I/O calls*, there is an opportunity to execute them in parallel. It also makes sure that those calls do not carry any dependence between them: not sharing any variables in the call arguments. Roslyn provides ready-to-use control & intra-procedural data flow analysis APIs that AsyncFixer uses to understand how variable flows in and out of regions of source.

Another challenge is to detect *async I/O calls*. To address this challenge, AsyncFixer uses a blacklist of all *async I/O calls* provided by .NET framework.

Fix. AsyncFixer does not *await* the consecutive *async I/O calls* so they only return tasks. Then, these tasks will be *awaited* together by using `Task.WhenAll()` so that multiple *async I/O calls* will be initiated in parallel. The results of these tasks will be gathered by using `Task.Result` expressions if necessary.

5.4.2 User defined async methods that are synchronously waited

Problem. `async/await` is a *"contagious"* programming model because asynchronous code tends to drive surrounding code to also be asynchronous. The code that calls an *async method* must not block waiting for the `Task` to complete, and so the caller will probably also `await` it. In this way, developers end up writing chains of *async methods*, each awaiting the next. This behavior is inherent in all types of asynchronous programming, not just the new `async/await` keywords.

Developers sometimes venture into asynchronous programming, converting just a small part of their app and wrapping it in a synchronous API so the rest of the app is isolated from the changes.

The *async method* (see lines 1-5 in Code 5.3) is called in line 9 and the returned `Task` is synchronously waited. A thread is blocked to wait for a `Task`, so such a valuable resource is wasted. This practice does not increase the performance or scalability in overall. Developers *await async methods* in order to benefit from them.

Code 5.3 Example for 5.4.2

```
1 async Task WriteAsync(byte[] message)
2 {
3     ...
4     await WriteToDatabaseAsync(...);
5 }
6 void OnNavigated(EventArgs event)
7 {
8     ...
9     WriteAsync(...).Wait();
10 }
```

Code 5.4 Fixed version of Code 5.3

```
1 async void OnNavigated(EventArgs event)
2 {
3     ...
4     await WriteAsync(...);
5 }
```

Detection. AsyncFixer finds all user-defined *async methods* (not provided by libraries) by checking if the method declarations have a `async` modifier. Then, for each async *async method*, it scans all call sites to check if the method is synchronously waited with `Task.Wait()` or `Task.Result` constructs.

However, there are some contexts where you cannot `await` *async methods*: query expression, in the catch or finally block of an exception handling statement, in the block of a lock statement, or in an unsafe context. If none of the call sites is in such contexts above, AsyncFixer reports those *async methods*.

Fix. Developer needs to manually fix this misuse. If *async method* needs to *awaited* in a call-site, the enclosing method of the call-site needs to be an *async method* as well because `await` keyword can only be used in *async methods*. Then, the developer needs to change the call sites of that enclosing method as well because `async/await` is a contagious model. Hence, AsyncFixer only reports those misuses, and the developer needs to fix the call site and its call graph.

5.4.3 Awaiting short running async calls

Problem. Although operations should be made asynchronous if they could take longer than 50 milliseconds to execute [61], developers can *overuse* `async/await`. For relatively fast async operations, the overhead may make asynchronous code less beneficial, particularly if many fast *async calls* need to be made frequently (e.g., in a loop). Adding `await` keyword comes at a price: the compiler generates some code that complicates the control flow, resulting in decreased performance.

Because AsyncFixer employs static analysis and it does not profile the programs, we decided to find the fast *async calls* by focusing on only one operation: `Task.Delay()`. This operation acts in a very different way than `Thread.Sleep()`. Developers use both of them to suspend the execution of a program (thread) for a given timespan. However, `Task.Delay()` will create a task which will complete after a time delay. If the time delay is less 50 ms (see line 7 in Code 5.5), this task causes some overhead, especially if `Task.Delay` is used in a loop.

Code 5.5 Real-world example for 5.4.3 from waslibs [62]

```
1 async void OnPointerWheelChanged(...)
2 {
3     ...
4     for (int n = 0; n < Math.Abs(offset); n++)
5     {
6         await TranslateDelta(delta);
7         await Task.Delay(10);
8     }
9 }
```

Code 5.6 Fixed version of Code 5.5

```
1 async void OnPointerWheelChanged(...)
2 {
3     ...
4     for (int n = 0; n < Math.Abs(offset); n++)
5     {
6         await TranslateDelta(delta);
7         Task.Delay(10).Wait();
8     }
```

```
9 }
```

Detection. AsyncFixer finds all *awaited* `Task.Delay()` calls whose delay argument that is less than 50ms.

Fix. AsyncFixer synchronously waits for those `Task.Delay()` calls instead of *awaiting* (see line 7 in Code 5.6). Hence, there will be no context-switching overhead which was previously caused by *awaiting*.

5.4.4 Unnecessary dispatcher to access the UI thread

Problem. Developers frequently call `Dispatcher.BeginInvoke()` method which allows developers to schedule the given action for execution in the UI thread. It is useful to update the UI after running long-running operation in the background. However, it is not needed if developers use `async/await` keywords. Because `await` captures information about the thread in which it is executed, the statements after `await` will be executed in the same thread. In Code 5.7, the statement in line 9 will be executed in the UI thread even without `BeginInvoke()`. In Code 5.7, the UI thread basically queues an action (statements between line 7-10) to herself (UI thread), resulting in unnecessary overhead.

Code 5.7 Real-world example for 5.4.4 from native-windows-phone-sdk [63]

```
1 async public Task LoadStoreOffers()
2 {
3     textField.Text = "loading";
4     var response = await Api.GetOfferListAsync(options);
5     Deployment.Current.Dispatcher.BeginInvoke(() =>
6     {
7         textField.Text = response;
8     });
9 }
```

Code 5.8 Fixed version of Code 5.7

```
1 async public Task LoadStoreOffers()
2 {
```

```
3      textField.Text = "loading";
4      ...
5      var response = await Api.GetOfferListAsync(options);
6      textField.Text = response;
7  }
```

Detection. AsyncFixer first locates the usages of `Dispatcher.BeginInvoke()` method. Then, it determines if the UI thread is responsible for execution. There are certain properties and methods that are only accessed by the UI thread: all members from `System.Windows` namespace. If the method has members from this namespace (see line 3 in Code 5.7), it means that the UI thread is responsible for the execution.

Fix. AsyncFixer first moves the statements in the lambda (see line 5-8 in Code 5.7) to the line before the dispatcher method. Then, it removes the dispatcher method call (see line 6 in Code 5.8).

5.5 Correctness Misuses

The misuses in this section are coding mistakes resulting in code that the developer did not intend.

5.5.1 Fire & Forget async calls in the `using` statement

Using statement. Before explaining this misuse, we should explain the `using` statement [64], which is a C# specific construct. It provides the syntax to specify the scope of the use of a resource object.

The `using` statement is useful for objects whose lifetimes are within the method or block in which they are created. This statement eliminates the need for multiple calls to release the resources at the end of their scope. For example, in Code 5.9, the `using` statement is initiated with a writer lock (line 1). Then, this writer lock will be automatically disposed and released at the end of block (line 5).

Problem. A `using` statement whose code block has some *fire & forget async calls* such as `PutBatchAsync()` in line 3 in Code 5.9 can result in a bug. When an *async call* is not waited or *awaited*, the lock might be released at the end of block before the async call completed. This interleaving will cause data races when accessing unprotected shared variables that are transitively reachable from `PutBatchAsync()`.

Code 5.9 Real-world example for 5.5.1 from NBitcoin [65]

```
1 using(lock.LockWrite())
2 {
3     InnerRepository.PutBatchAsync(...);
4     ...
5 }
```

Code 5.10 Fixed version of Code 5.9

```
1 using(lock.LockWrite())
2 {
3     await InnerRepository.PutBatchAsync(...);
4     ...
5 }
```

In industry code, we have seen many instances of this misuse. For instance, the `using` statement is commonly used to measure the execution time of a code block. However, when there are *fire & forget async calls* in the block, they are not accounted for in the block's execution time, causing misleading performance measurements.

Detection. AsyncFixer scans each top-level method call in the `using` statements to check if it is a *fire & forget async call*. It does not scan the method calls inside the call graphs.

Fix. AsyncFixer inserts the `await` keyword to asynchronously wait the *fire & forget async call* if the enclosing method definition is an `async` method (see line 3 in Code 5.10). Otherwise, the *async call* will be synchronously waited such as `asyncCall.Wait()`.

5.5.2 Fire & forget Task.Delay calls

Problem. Some *fire & forget async calls* methods need to be *awaited* or waited. For example, when the async `Task.Delay()` is not *awaited* or waited, no delay will be inserted between `Task.Delay` statement and the next statement, causing erroneous behavior.

In Code 5.11, the developer intends to insert quarter-second delays between `SetFrameState()` calls. However, the developer forgot to *await* or wait the `Task.Delay()`, so this statement has no effect.

Because this misuse is so common in the open-source programs, we contacted several developers. Most of them did not notice that `Task.Delay()` is an asynchronous call which returns a `Task`. They thought that this operation is a direct replacement of the `Thread.Sleep()` method (see Section 5.4.3 for more information).

Code 5.11 Real-world example for 5.5.2 from Windows10IoTAdaFruitLed [66]

```
1 for (byte l = 0; l < 2; l++)
2 {
3     matrix.SetFrameState(LedDriver.Display.Off);
4     Task.Delay(250);
5     matrix.SetFrameState(LedDriver.Display.On);
6     Task.Delay(250);
7 }
```

Code 5.12 Fixed version of Code 5.11

```
1 for (byte l = 0; l < 2; l++)
2 {
3     matrix.SetFrameState(LedDriver.Display.Off);
4     await Task.Delay(250);
5     matrix.SetFrameState(LedDriver.Display.On);
6     await Task.Delay(250);
7 }
```

Detection. AsyncFixer scans all `Task.Delay()` calls to check if the returned `Task` is *awaited* or waited. The challenge is that there are many constructs to synchronously wait a `Task`: `Task.Wait()`, `Task.WaitAll()`, `Task.GetAwaiter().GetResult()`. Developers sometimes set a variable to the returned `Task` and then, this variable is *awaited* or waited

with one of the constructs above. Hence, AsyncFixer employs data-flow analysis to track the `Task`.

Fix. AsyncFixer *awaits* or synchronously waits the `Task.Delay()` call based on whether the enclosing method is an *async method*. AsyncFixer inserted `await` keywords in front of `Task.Delay` calls in Code 5.12 because the enclosing method is an *async method*.

5.5.3 Awaited async call in the compound assignment

Problem. The *pause 'n' play* semantics featured in `async/await` has many implications. It might cause interleavings that can be the source of timing-related bugs. Such a subtle bug happens when developers combine the `await` keyword with compound assignments (e.g., +=, *=, &=).

In Code 5.13, the C# compiler will rewrite the statement in line 6 into the following:

```
total = total + await provider.Ask(...);
```

This statement executes in the following (non-atomic) steps: (1) load `total` onto the stack, (2) *await* the task, (3) push result of task onto the stack, (4) add the stack values, (5) store into `total`.

Suppose that `provider.Ask()` is a long-running operation, causing the execution to pause at step 2. Because `total` variable is an instance variable, some other threads might modify this variable in the meantime. When `provider.Ask()` completes, its result will be pushed onto the stack and the values in the stack will be summed up. However, the stack had the old value for `total`. Because step 1 has already executed, the `total` is now stale.

This timing-related is not one of the usual data-races because this can happen in a single-threaded UI environment as well. Imagine that `Search()` method is called when the user clicks a button. If the user clicks the button again before the completion of first click, the event handler in the second click will read the stale value. This subtle bug is a data-race specific to `async/await`, that cannot be detected by existing race-checker tools.

Code 5.13 Real-world example for 5.5.3 from Orleankka [67]

```
1 int total;
```

```
2 async Task Search(string api)
3 {
4     ...
5     total += await provider.Ask(new Search(query));
6     ...
7 }
```

Code 5.14 Fixed version of Code 5.13

```
1 int total;
2 async Task Search(string api)
3 {
4     ...
5     int temp = await provider.Ask(new Search(query));
6     total += temp;
7     ...
8 }
```

Detection. AsyncFixer finds all compound assignment statements whose right-hand side expressions contain `await`. Then, it determines whether the left-hand side expression is a shared variable (i.e., not a local variable).

Fix. To avoid this misuse, developers should not mix `await` and the compound assignment operators. AsyncFixer replaces the compound assignment statement with two different statements: (1) the first statement stores the result of the *awaited* `Task` (see line 5 in Code 5.14), (2) a new compound assignment that uses the variable in the previous statement (see line 6).

5.5.4 Implicit down-casting from `Task<Task>` to `Task`

Problem. `Task.Factory.StartNew()` is the primary method for scheduling a new `Task` in .NET. Developers usually mix this method call with `async` lambdas that are basically anonymous methods returning a `Task`. This practice can backfire in some cases.

In Code 5.15, the argument of `StartNew()` method call is an `async` lambda (see lines 3-10). `StartNew()` returns a `Task<T>`, where `T` is the return type of the lambda argument. In

Code 5.15, it returns a task within another task, `Task<Task>` because of the async lambda's return type.

In line 3, the `await` statement will pause the execution until the outer task completes. The outer task represents only the first part of the async lambda: the execution until the `await` point in line 5. Hence, the return statement in line 11 will be concurrently executed with the second part of the async lambda (see lines 6-9). The developer erroneously think that the second part of lambda and the return statement run sequentially; however, it runs concurrently; this causes data-races and unintended behaviour.

There is an important danger in Code 5.15: a silent exception. If `DeAuthenticate()` call returns false, an exception in line 8 will be thrown to the inner task. The exceptions in a task can only propagate to the caller if the task is *awaited* or waited. In line 3, only the outer task is *awaited* so the exceptions in the inner task will be swallowed and have no effect at all: the user will never see the "DeAuthenticate" error and the developer cannot handle this exception.

Code 5.15 Real-world example for 5.5.4 from app-crm [68]

```
1  public async Task<bool> LogoutAsync()
2  {
3      await Task.Factory.StartNew(async () =>
4      {
5          var success = await DeAuthenticate();
6          if (!success)
7          {
8              throw new Exception("Failed DeAuthenticate");
9          }
10     });
11     return true;
12 }
```

Code 5.16 Fixed version of Code 5.15

```
1  public async Task<bool> LogoutAsync()
2  {
3      await Task.Run(...);
4      return true;
5  }
```

Detection. AsyncFixer finds `StartNew()` calls that take async lambdas as an argument. Among these calls, it reports the ones that (i) are not double *awaited* and (ii) are not followed by `UnWrap()` call.

Fix. `Task.Factory.StartNew()` does not understand async lambdas. `Task<Task>` is implicitly down-casted to `Task`, so the inner task cannot be *awaited* or waited. There are several ways to fix this problem. Developers can double await the nested task:

```
await await Task.Factory.StartNew(async ...)
```

AsyncFixer uses another approach to fix it: it replaces `Task.Factory.StartNew()` with `Task.Run()` like in Code 5.16. `Task.Run()` automatically unwraps the nested task and returns the inner task.

5.6 Bad Practices

The misuses in this sections are less severe, anti-patterns which are violations of recommended and essential coding practices.

5.6.1 AggregateException when synchronously waiting for async calls

Problem. Developers should avoid synchronously waiting for *async calls* (see Section 5.4.2). However, there are cases when developers cannot *await async calls* such as inside a `lock` statement, or inside the entry method of the program (main method). In such cases, developers wait for *async calls* by using `Task.Result` or `Task.Wait()` methods. These methods always encapsulate the inner exception in an `AggregateException`. This type of exception makes the exception handling code more complex. The debugging experience is not preferred as well because the exception message is generic: *"One or more errors occurred"*.

Detection. AsyncFixer reports *async calls* which are waited with `Task.Wait()`. It also reports *async calls* which are not *awaited* and whose results are accessed by `Task.Result` expressions.

Fix. AsyncFixer replaces the `Task.Wait()` or `Task.Result()` with `GetAwaiter().GetResult()`. This new method preserves the stack trace and task exceptions instead of wrapping them in an `AggregateException`, providing simplified post-mortem debugging.

5.6.2 Fire & forget tasks

Problem. Exceptions thrown in a task are automatically captured, and then re-thrown when the task is (i) *awaited*, (ii) synchronously waited, or (iii) passed to another task via continuation. If none of these three conditions is met, uncaught exceptions in task are silenced. Hence, if developers use fire & forget tasks, they can never handle and know the exceptions occurred in those tasks.

Detection. AsyncFixer scans all task creation calls (e.g., `Task.Run`, `TaskFactory.StartNew`) and figures out if they are *awaited*, waited, or chained with some other tasks. Otherwise, AsyncFixer reports those task creation calls as fire & forget tasks.

Fix. AsyncFixer inserts the `await` keyword to asynchronously wait the task if the enclosing method definition is an `async` method. Otherwise, the fire & forget task will be synchronously waited such as `task.Wait()`.

5.7 Evaluation

To evaluate AsyncFixer, we answer the following research questions:

EQ1: *How applicable is* AsyncFixer*?*

EQ2: *Are the program transformations safe?*

EQ3: *Is* AsyncFixer *fast enough to be used interactively?*

EQ4: *Do developers find these transformations useful?*

5.7.1 Experimental setup

We evaluated AsyncFixer by running it in batch mode over both open-source and proprietary software.

Open-source corpus. We gathered open-source C# programs from Github. We filtered the programs that have at least one commit since January 1, 2016 because we want to analyze recently updated programs. Then, we sorted them by popularity and gathered 500 of them. In total, the corpus comprises 24M SLOC, spanning a wide spectrum from web & desktop programs to libraries and mobile apps.

Proprietary corpus. To gather the industry data, we conducted extensive visits at two companies. The first company is a leading company that provides health information services. We analyzed C# projects that target library, console, and ASP.NET (web) platforms, comprising 2.5M SLOC.

As a second opportunity for the industry data, we visited a global technology company where we analyzed the C# code-base of the devops tools. We analyzed 1.5M SLOC that target library and console platforms.

In total, we analyzed 4M SLOC to gather statistics from the proprietary corpus.

5.7.2 Applicability

We ran AsyncFixer in batch mode over the entire corpus. We counted the number and kinds of misuses that it detected.

Table 5.2 tabulates the number of misuses for each kind separately. In total, AsyncFixer detected 891 in open-source corpus and 169 in proprietary corpus. As shown in the third column, there are no numbers for some kinds of misuses, because AsyncFixer did not have the detection of some kinds of misuses when we visited the first company.

It is surprising that the density of misuses in the proprietary corpus is higher than in the open-source corpus. We noticed that developers synchronously wait *async methods* as the most common misuse in both open-source and proprietary corpus.

Table 5.2: Misuse statistics of `async/await`. In the second column, misuse type is represented by the section number. The third column shows the results for open-source corpus. The fourth and fifth columns show the results for proprietary corpus.

Category	Type	Open-source	Industry #1	Industry #2
		# misuses	# misuses	# misuses
Performance	5.4.1	63	-	12
	5.4.2	422	25	32
	5.4.3	52	-	6
	5.4.4	35	2	0
Correctness	5.5.1	34	-	26
	5.5.2	32	0	0
	5.5.3	19	0	3
	5.5.4	29	-	11
Bad-practices	5.6.1	120	10	24
	5.6.2	85	0	18

5.7.3 Safe transformations

We checked the safety of AsyncFixer transformations in two ways. First, when running AsyncFixer over the code corpus, it also fixed the found misuses. Our evaluation script compiled the program in-memory and determined that no compilation errors were introduced. Second, we sampled and manually checked 10% of the fixed instances in the open-source corpus and *all* the fixed instances in the industry corpus. We determined that the fixes are correct. We also sent the fixes as patches to developers (see Section 5.7.5).

5.7.4 Performance

We measured the execution time of AsyncFixer in batch mode. For 100K SLOC, AsyncFixer took 1.3 minutes with all misuse detections enabled. However, the performance depends on many other factors besides #SLOC: the number of async methods, call graph depth of async methods, etc.

AsyncFixer is also fast enough to be used as a live analyzer in Visual Studio. Because it runs on a background thread to analyze the code-base of the active project in the IDE, developers do not experience any slowdown.

5.7.5 Developer feedback

Our industrial partners found AsyncFixer useful. They integrated AsyncFixer directly into their automated build process. When developers build their code-base from Visual Studio or command-line, they see the misuse warnings reported by AsyncFixer in the build output. They also used AsyncFixer to fix the reported misuses in Table 5.2.

To further evaluate the usefulness in practice, we submitted (via pull requests) 56 patches generated by AsyncFixer in 20 recently updated open-source programs. Developers accepted patches for 41 misuses in 15 programs. The developers of 4 programs did not reply and the developers of one program did not like the fix for Section 5.5.1.

We have received very positive feedback. The developer of oscill [69] said *"idea is great. It's proof-of-concept work and it fully worked"*. One developer from CodeSmithTools [70] *"love(s) the* AsyncFixer *package"*, and he found it *"extremely useful"*.

AsyncFixer also educates developers. The developer of languagedetectapp [60] said *"your patch has updated my knowledge about running multi-task, thanks so much"*. The developer of nbitcoin [65] was not aware about the subtle misuse that AsyncFixer found in his code and said *"very nice catch"*.

5.8 Discussion

Our previous study [15] already influenced the design of the C# compiler. We previously reported a very common anti-pattern, when developers unnecessarily use `async/await`. Based on our empirical data, Neal Gafter, the main C# compiler architect, proposed a specific compiler optimization for this anti-pattern [71]. According to the proposed optimization, if the C# compiler detects the unnecessary usage, it will not generate the state machine and ignore `async/await`. This optimization is scheduled to become part of C# 7. Similarly, we

expect that our current study will have a significant impact on future versions of C#.

Other major programming languages (e.g., Java) have plans to introduce similar constructs for asynchronous programming. Our findings can guide language designers to an improved design of `async/await` for their languages.

We have seen a wide variety of misuses of the `async/await` keywords in our code corpus. Different authors, both from Microsoft and others, have documented these potential misuses extensively. We have noticed that the samples and tutorials in Github have misuses as well. If developers start to learn `async/await` from flawed tutorials, the misuses get ingrained into the developers' culture, they are hard to un-learn. Hence, to educate developers, we provide the misuse examples in our website: `http://LearnAsync.NET`. We believe that documentation and tool support are complementary, and should both be provided.

5.8.1 Threats to validity

External. For what C# programs are the results representative? We analyzed both industry and open-source code-bases. Open-source programs span a wide domain, from games, social networking, and office productivity to image processing and third party libraries. They are developed by different teams from a large and varied community. Our industry code-base consists of reusable libraries, server side code (ASP.NET), and console programs, developed by approximately 150 developers.

Internal. Our anti-pattern analysis is an under-estimation: For example, we detect one form of *avoid short running async operations*, but likely there are other fast operations that should not be async'ed. In other cases, heuristics are used, for example to discover the existence of short running operations, or for detecting UI context. So we can conclude that the results are a lower bound on misuse numbers, and are likely to be slightly higher.

5.9 Summary

Developers fully embraced the novel `async/await` across all domains. However, we discovered 10 kinds of prevalent misuses that have severe consequences: subtle data-races, swallowed exceptions, and unintended behavior. These are a new source of bugs in today's programs. Because no other tool can detect these bugs, we designed, implemented, and evaluated a new static analysis tool, AsyncFixer that (i) detects and (ii) fixes `async/await` misuses.

Our large-scale empirical evaluation shows that AsyncFixer is highly applicable: it discovered 891 misuses in open-source and 169 misuses in proprietary code-bases. Our industrial partners have already started to use AsyncFixer in their automated build process. We hope that our work serves as a call to action for the testing, program verification, and program analysis communities to tackle an increasingly important source of bugs.

CHAPTER 6

A Guide for Researchers Studying New Constructs

Programming languages and standard libraries evolve so that they are more powerful and expressive, leading to more sophisticated software. Technology trends drive this evolution and decide new constructs. For instance, the competitive mobile app market has led to fast & responsive user interfaces, which require asynchronous programming. Programming languages and standard libraries provided new constructs (e.g., `async/await` keywords) to treat asynchrony as a first-class citizen.

In future, new programming constructs will be released in a similar way for trends stemming from other needs, e.g., to process big-data. We expect that these new constructs will be released fast without adequate tool support and extensive empirical & usability testing. Subsequently, researchers can fill these gaps by empirically studying these new constructs and providing necessary tools to help developers adopt them. Here, we provide a guide for researchers studying new constructs with the goal of producing a practical impact.

We have followed ourselves the principles outlined in this guide and found them to hold in each of the three pieces of technical work that compass this dissertation. Figure 6.1 shows the interaction among the five steps of our guideline.

6.1 Choosing a Construct to Study

The first step is to choose a new construct to empirically study. A programming construct is either a language keyword or an API class from the standard library. Because we need to analyze the usages of this new construct, we selected constructs for which there exist instances in real-world software. When we first studied `async` keywords in 2013, they were one-year old. However, we were able to find many usages in the Windows Phone applications

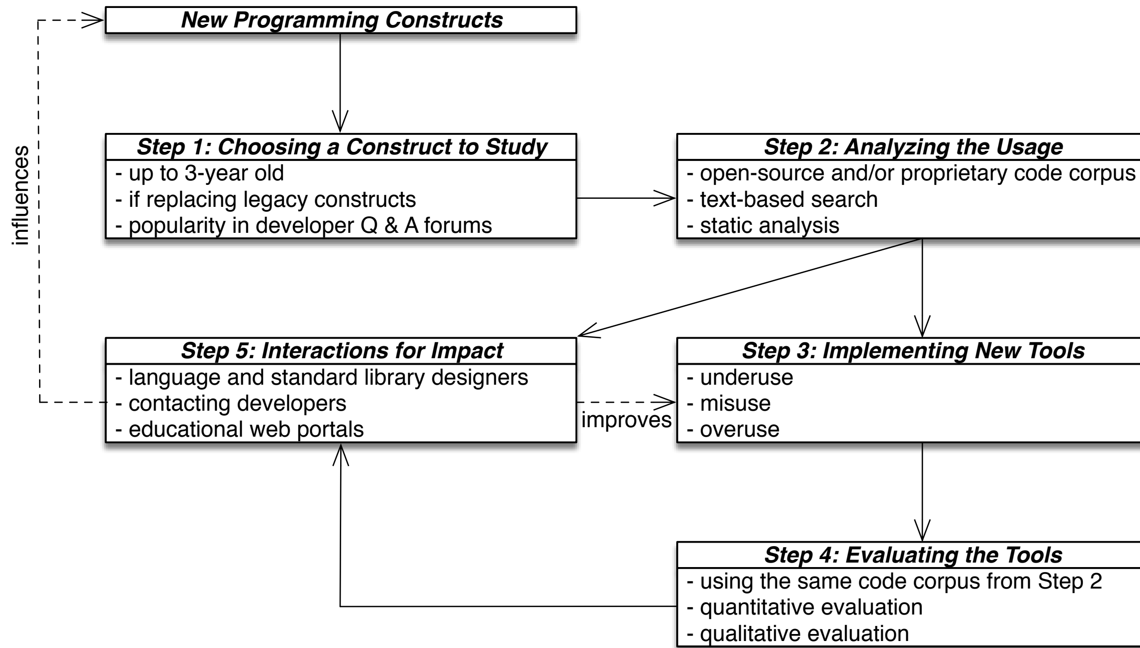

Figure 6.1: Guide for researchers studying new programming constructs

because of the beta release of those keywords in 2011. Hence, to maximize the likelihood of finding instances, we studied constructs which are up to 3-year old (including the first beta version).

To maximize the likelihood of practical impact, it is best to choose a new construct which can replace the usages of some existing constructs and thus require developers to migrate their legacy code. For instance, `async` replaces call-back asynchronous constructs, `Parallel` class replaces fork & join `Task` instances. C# 7 will introduce pattern matching soon, which will replace specific `switch` statements. These new constructs are valuable for researchers because developers desperately need automated migration and refactoring tools for upgrading their codebases.

To find inspiration, we also recommend to look at developer Q & A forums such as Stack Overflow [35]. These forums can highlight which are the constructs that developers find hard to use or error-prone.

6.2 Analyzing the Usage from Early Adopters

We analyzed the real usages of the new construct to check whether there were usability problems. Because Github [16] is currently the most popular code repository hosting service, it is the best place for researchers to mine the usages of the construct. We first searched the construct by using the basic search functionality in Github's webpage. The text-based search results showed us in which contexts the construct was commonly used. These results helped us develop the static analysis that accurately detects the usage of the construct.

There are choices of static analysis frameworks for each major programming language: e.g., WALA [107] and Eclipse JDT [108] for Java, Clang [109] and Eclipse CDT [110] for C++, Roslyn [18] for C#. Researchers can use these frameworks to build static analysis tools that reveal more accurate information about real-world usages of the construct. We started the implementation with over-approximation, allowing many false-positives and almost-zero false-negatives. Then, we improved the accuracy of the static analysis to tolerate fewer false-positives.

After we implemented an initial version of the static analysis tool for detection, we downloaded the code repositories from Github to run the detection tool. Based on the text-based search, we had an idea in which type of software the new construct was likely to be used. For instance, when we first studied `async`, we only downloaded Windows Phone applications from Github because these constructs were not popular outside the mobile domain at that time. Hence, we downloaded all C# code repositories and then filtered the repositories that contain a specific configuration file which was specific to Windows Phone.

When setting up the corpus of data, the goal is to do filtering as much as possible before downloading the repositories. Github rest API [111] allows researchers to find repositories via various criteria: size, date of creation, when they were last updated, language they are written in, the number of stars, keyword which will be searched in the description of the repository. We detected repositories which were last updated after the release date of the construct to be studied. As another criteria, we ignored the forked repositories. In the Git community, developers often fork a repository and start making changes in their own copies. Sometimes, the main repository might merge changes from the forked repositories, but many

times the forked repositories start evolving independently. Because these forked repositories share most of their codebases, it is best to ignore those and include only the main repository. Then, we downloaded the most popular repositories (i.e., sorted by the number of stars) that match our criteria.

After downloading repositories, we eliminated two types of repositories as well to increase the quality of their code corpus. First, we eliminated the "toy repositories", e.g., the ones that have less than 1000 non-comment, non-blank lines of code (SLOC) because many were just experimentally written by developers who learn a new construct, and they did not represent realistic usage of production code. Second, we eliminated the repositories which did not compile due to the missing libraries, incorrect configurations, etc., since these repositories would not be processed correctly by static analysis tools.

6.3 Implementing the Tools that Developers Need

The most important step is to understand what kind of tool support developers need regarding the new construct. This decision should be data-driven, based on the previous step. We also contacted the developers already using the new construct to understand if they need tool support. For example, in our ASE-2015 paper [112] we contacted developers but also technical authors who blog about this new construct.

We focused on three problems that require the creation of new tools: (i) underuse, (ii) misuse, (iii) overuse. We aimed to solve these problems with automated program transformation tools.

6.3.1 Underuse

We found that developers are not using the new constructs due to two different reasons. First, they are not aware of them. Second, they are aware of them but they find it tedious to refactor their existing (legacy) code. Because these two reasons require different types of tool solutions, it is valuable for researchers to distinguish between them based on the usage data.

If the new construct is (almost) always used in the code that has been created after the release date of the construct, it means that developers are aware of the construct and they prefer using it in their new code rather than the existing code. In this scenario, developers know where to introduce the new construct but they find the migration tedious and/or challenging. Hence, researchers should consider developing an automated migration tool for legacy code.

If the construct is rarely used in the recent code, researchers should consider developing a transformation tool suggesting code snippets that can be transformed to the new construct. This recommendation workflow can make developers aware of the new construct.

6.3.2 Misuse

We found that developers usually misuse the new constructs due to several reasons. To find out the misuses, researchers should start with looking at public bug reports in Github. Github webpage allows users to search a specific keyword in the description of the bug reports (i.e., issue in git jargon). Hence, researchers can search the name of the construct in the bug reports and find the misused patterns.

Second, they can manually compare the real-world usages with the guideline that the language designer publishes regarding the new construct. Programming language designers usually write a guideline about how to use the newly released constructs. This guideline can be useful to find out the misused patterns in the real-world usages.

6.3.3 Overuse

We found that when developers start using a new construct, they sometimes overuse it. They introduce the new construct to some code which does not benefit from it at all. In this case, the construct can even degrade the code performance or readability. Hence, researchers should manually review the real-world usages to detect in which contexts the new construct is unnecessarily used. If necessary, researchers contact the developers of those usage instances to confirm whether developers really need to use the construct in those contexts.

6.4 Evaluating the Tools

We extensively evaluated the usefulness of program transformation tools that help developers adopt the new construct. In our studies, we conducted two kinds of empirical evaluation: *quantitative* and *qualitative*. For both evaluation methods, we used the same code corpus that we prepared from Github (see Section 6.2).

6.4.1 Quantitative evaluation

To quantitatively evaluate the usefulness of the program transformation tool, we answered questions like the ones below:

1. How applicable is the tool? To evaluate the applicability, we executed the tool over the code corpus. We measured how many times the tool was applied and the number of false-positives and false-negatives. We discussed the soundness and completeness of the tool under this question.

2. Are the automated transformations safe? It is important to evaluate the safety of transformations applied by the tool. After each transformation, it should be automatically checked whether any of them break the existing unit test suite or the software with a compilation error. We also randomly sampled and manually checked some of transformed instances.

3. Is the tool fast enough to be used interactively? The program transformation tool is useful if developers can use it interactively in their IDE. Hence, it is important for developers if the tool detects the code snippet and applies the transformation in a fast manner. The detection and the transformation of one instance should be measured all together. Because developers sometimes want to run the automated tool against their projects in batch mode, it is valuable to measure how much time the tool takes on average to run against a project.

4. How much programmer effort is saved by the tool? The programmer effort is directly correlated with the size of the changes. We evaluated the impact of the program

transformations by investigating the size of the changes. We measured how many lines of code were changed per transformation by the tool.

6.4.2 Qualitative evaluation

We further evaluated the usefulness of the tools in practice. Because it is hard to acquire real users of the tool in the experimental version, we applied the tool ourselves, and offered the transformations to the original developers as a patch via a pull request. We also asked developers whether they need such a tool to automate those transformations.

To increase the response rate for the pull requests, we identified recently updated repositories and offered patches to a limited number of those repositories using an iterative process: first experiment with the length of the pull request, the recency of the code affected by the pull request. These experiments will teach the researcher how to create pull requests that have a higher chance of being reviewed by open-source developers.

6.5 Interactions for Impact

To increase the impact of our research, we constantly communicated with (i) language and standard library designers and (ii) developers.

6.5.1 Language and standard library designers

The empirical evidence from Section 6.2 determines which tool support is appropriate, but can also have a tremendous impact on the design of the construct. For researchers who are interested on having practical impact on the language constructs, we recommend going the extra mile. We shared our empirical findings with the designers of the construct in C#. We did not only inform them of our findings but also suggested solutions in terms of language design. Our fruitful collaboration resulted in a new C# compiler optimization that eliminates a common misuse.

We believed that our findings can cross language boundaries because programming lan-

guages release the same construct due to competition and technology trends. Hence, we provided our findings to the designers of the construct in other languages such as F#. They appreciated our effort and claimed that they would use our findings for language revisions. We also contacted the designers of the languages that consider adding the same construct in the future, for example Scala. Because it is easier to change the proposed design before it reaches production, we thought that we can have a more impact on those languages. For instance, `async/await` keywords might not capture the threading context by default in other languages, based on our suggestion.

Our communication with the language designers was not one-way. They often shared their reasonings about the findings and suggested new research questions to study. Hence, our empirical studies have been enriched by our communication with the construct designers.

6.5.2 Developers

Throughout our research, we contacted developers many times to ask questions and feedback. For a widespread impact, we aimed to both (i) understand their needs and (ii) educate them with our findings. Based on our findings and communication with developers, we noticed that (i) developers miss real-world usage examples of the construct and (ii) even the official samples and tutorials might contain misuses. Hence, we designed educational web portals (`http://LearnAsync.NET` and `http://LearnParallelism.NET`) to share our empirical findings with developers. These educational resources attracted more than 200,000 of visitors from 141 countries, showing that developers need real-world code examples.

We did not only educate the developers through our web portals but also acquired early adopter users for our tools. When we listed thousands of real-world misused examples on the webpage, we put a sign-up form for them to register and get notified when we release our tools. Thus, we first released our tools to these early adopters from our mailing list (around a thousand early adopters) and got early-feedback. More importantly, they helped us improve the accuracy of our static analyses by reporting corner-cases and false-positives that they discovered. We noticed that those early adopters were generous with their time

and explained the false-positives with examples. Because they value the tools that save them a lot of time, these early adopters contributed back to these tools.

CHAPTER 7

Related Work

We briefly present the work that is closest to the dissertation. We organize the related work into: (i) empirical studies for language and API usage; (ii) empirical studies for concurrency bugs; (iii) concurrency misuse detection; (iv) refactoring for Concurrency.

7.1 Empirical Studies for Language and API Usage

There is a multitude of documented empirical studies into the use of language constructs or libraries. These range from deep investigations into specific projects to very broad surveys of huge numbers of projects.

Some small-scale studies include Karus et al.'s "[A] Study of Language Usage Evolution in Open Source Software" [72], which describes how developers combine use of different languages and artifacts in open source projects. Parnin et al. [73] studied the adoption patterns of Java generics in open-source applications. Hoppe and Hanenberg [74] also performed a small empirical study to determine if generic types in Java provide benefit to developers.

There are several large-scale studies into the use of programming languages. Callau et al. [75] investigated into 1000 Smalltalk projects of "How (and why) developers use the dynamic features of programming languages [...]". Grechanik et al. [76] analyzed 2080 Java projects to understand how general language features are used. Torres et al. [7] conducted a study on the usage of concurrent programming constructs in Java, by analyzing around 2000 applications. Dyer et al. [77] analyzed 31k open-source Java projects to find uses of new Java language features over time. Buse et al. [78] proposed an automatic technique for synthesizing API usage examples and conducted a study on the generated examples. Pankratius et al. [79] studied the current state-of-the-practice in shared-memory, multicore

programming, and suggested areas and engineering principles for future research. Others [80] have studied the correlation between usage of the MPI parallel library and productivity of the developers.

Weslley et al. [7] studied how developers are retrofitting applications to become more concurrent and summarized the usage of concurrent programming constructs in Java by analyzing more than 2000 projects. They give some coarse-grain usage results like the number of synchronized blocks and the number of classes extending `Thread`. In contrast, our studies look at every concurrency construct in the concurrency libraries, and we also look at how these constructs form patterns and structures. Although they analyze the usage of very few constructs, their results are not accurate due to missing type information because they only perform lexical analysis. Also, their count of the constructs' usage can be misleading. For example, they measure the usage of `java.util.concurrent` by counting statements that import the library. In our study, there are many applications that import the `Task` library but never invoke any constructs. For example, there is an application, *DotNetWebToolkit* [81], that imports TPL 111 times but invokes TPL just once. To best of our knowledge, our empirical studies are the first large-scale studies that uses both syntactic and semantic analysis, thus increasing the accuracy of the usage statistics.

Monperrus et al. [82] study the API documentation of several libraries and propose a set of 23 guidelines for writing effective API documentation. Robillard and DeLine [58] study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples. Our studies provide educational portals which contain lots of usage examples from real code which can hopefully educate newcomers to concurrency libraries.

These studies cover a wide range of topics. However, we are not aware of large-scale empirical studies into the use of language constructs or libraries for specifically parallel and asynchronous programming for C#.

7.2 Empirical Studies for Concurrency Bugs

Many works focus on studying concurrency bugs. Lu et al. [83] categorized concurrency bug types by analyzing a large number of bug reports from open-source repositories. In

a followup work [84] they also described bugs that manifest as performance slowdowns in concurrent programs. Schaefer et al. [85] showed several examples of how sequential refactorings can break concurrent programs. Fonseca et al. [86] presented a study of concurrency bugs in MySQL, in which they categorized and analyzed the effects (e.g., program crash, assertion violation) of the concurrent bugs. Another study [87] focused on identifying the contented resources (e.g., shared cache) that adversely impact the parallel performance. Li et al. [88] studied and categorized bug characteristics in modern software. Their result shows concurrency and performance related bugs can have a severe impact on software.

Dig et al. [89] studied the evolution of Java concurrent applications and cataloged the changes that programmers made in response to concurrency. Another work [90] on automated refactoring to introduce concurrent library constructs shows that manual refactorings from `HashMap` to `ConcurrentHashMap` are error-prone.

To best of our knowledge, there is no study that looked at the misuses of `async/await` keywords.

7.3 Concurrency Misuse Detection

There are some recent work focusing on the misuse detection for concurrency. Lin and Dig [91] recently mined a corpus of 28 widely used open source Java projects for violations of check-then-act misuse of concurrent collections. Like our work, their work documents a set of anti-patterns, which they hope will inform library designers to build more resilient APIs.

De Wael et al. [92] studied a corpus of 120 open source Java projects that make use of the Fork/Join framework. They documented three common anti-patterns in this framework and studied the impact on performance of these anti-patterns. Unlike our study, they did not implement a tool to detect these anti-patterns in source-code.

Pattern inference and identification is also a widely used approach to improve software quality. `AVIO` [93] and `Falcon` [94] analyze the access patterns of variables to detect or locate concurrency bugs. `FindBugs` [95] detects bugs by statically matching the bug patterns to programs. Yu et al. [96] exploited interleaving idioms to test concurrent programs. Uddin

et al. [97] inferred temporal API usage patterns that can be used to improve the API design and usage. Wendehals and Orso [98] proposed a dynamic technique to recognize design patterns in the programs.

The success of the previous technique in finding buggy idioms shows that despite the fact that the buggy idioms are not particularly deep, today's state-of-the-art systems are still riffle with such bugs. Thus, custom pattern-based analyses, like our current work on patterns of `async/await` usage, can be quite effective.

To best of our knowledge, there is no anti-pattern detection for asynchronous programming and specifically for `async` keywords.

7.4 Refactoring for Concurrency

There have been numerous publications of studies on refactoring since Griswold first published a doctoral dissertation on the subject [99], with various goals. Of primary interest to us is refactoring for concurrency. The introduction of concurrency in programs can be categorized to have four different objectives [89]:

Latency. Latency is the subject of this thesis: it measures the time it takes for an inquiry to return an (initial) response. Heavy operations in the UI thread increase latency, causing responsiveness problems. Lin et al. [100] developed a refactoring tool to extract heavy operations in the UI thread to asynchronous execution.

Throughput. Task and data parallelization can improve the number of work units executed per time unit. This form of concurrency has evident use: it decreases the time the user has to wait for an inquiry to return its final result.

Java 8 introduced bulk data manipulation support, both for sequential processing and for parallel processing. A first step in moving to parallel data processing from sequential data processing through refactoring is by transforming traditional for-loops to the new bulk data APIs as described by Gyori et al. in [101].

Another example is the ReLooper tool by Dig et al. [102] which can automatically refactor arrays into `ParallelArrays` to do parallel processing of the data in the array.

Scalability. Application performance should scale up with the number of execution units that it runs on. This is directly related to Amdahl's law [103] by decreasing the fraction of computation that must be executed sequentially, total performance can be increased. Generally speaking, by using the right data structures, the scalability of programs can be improved. For example, Dig et al. [90] published a study on how code using Java HashMaps can be refactored to take advantage of Java 5's ConcurrentHashMaps, thereby enabling more code to be executed in parallel.

Correctness. It is assumed that code is functionally correct for sequential execution. However, that does not mean that the same code is thread-safe. The code must then be transformed into a thread-safe form, before introducing any kind of concurrency.

To the best of our knowledge, there is no refactoring tool that specifically targets asynchronous programming. In industry, ReSharper [104] is a well-known refactoring tool, but it does not support `async`-specific refactorings. ASYNCIFIER helps developer design responsive apps, which is an area that has not yet been explored.

None of these previous tools address the problem of migrating between different levels of abstractions in (already) parallel code. Balaban et al. [105] present a tool for converting between obsolete classes and their modern replacements. The developer specifies a mapping between the old APIs and the new APIs. Then, the tool uses a type-constraint analysis to determine if it can replace all usages of the obsolete class. Their tool supports a one-to-one transformation whereas SIMPLIFIER supports many-to-one transformations. Even our one-to-one transformations from TASKIFIER require custom program analysis, e.g., detecting I/O blocking operations, and cannot be simply converted by a mapping program.

CHAPTER 8

Conclusion and Future Work

8.1 Revisiting Thesis Statement

It is inevitable for developers to use concurrency in their software due to the need for responsiveness and performance. Hence, programming language and library designers treat concurrency as a first-class citizen. They provide faster, more scalable, and more readable concurrency constructs every other year. However, they do not check how developers embrace these new constructs and if developers are in trouble with using them. We claim that concurrency constructs deserve first-class citizenship in empirical research and tool support, too.

Because our thesis statement is three-pronged, we would like to separately visit each of our claims:

(1) Empirical studies about concurrency constructs can provide insights into how developers use, misuse, or underuse concurrency.

Our empirical studies [6, 15, 19, 112] show that developers still use old (low-level) concurrency constructs. When developers use concurrency for parallelism, heavyweight `Thread` and `ThreadPool` constructs are still the primary choices for them. They are also oblivious to the benefits brought by the higher-level concurrency constructs such as `Parallel` and `Task`.

When developers use concurrency for asynchrony, they heavily use callback-based concurrency constructs that invert the control flow and obfuscate the intent of the original synchronous code. When they start using new `async/await` constructs, they either overuse or misuse them. We found that 14% of methods that use `async/await` do not need to be `async` at all. Developers also misuse `async/await` in many different ways. These async/await

misuses have severe consequences. Some significantly degrade the performance of async programs or can even deadlock the program. Others cause subtle data-races that are unique to `async/await` and cannot be detected by previous race detecting techniques. Other misuses "swallow" the run-time uncaught exceptions thrown by a program which hinder developers when searching for errors.

(2) It is possible to design and develop interactive program transformations that enable developers (i) to migrate their software to modern concurrency constructs and (ii) to fix misused modern constructs.

Library and programming language designers expect developers to manually migrate the legacy software to modern concurrency constructs. However, the manual migration is tedious and has several challenges. This migration cannot be done by a simple find-and-replace tool, but it requires custom program analysis and transformation. We hypothesized that interactive program transformation tools can mitigate these problems. Hence, we present three interactive refactoring tools:

TASKIFIER safely transforms old style `Thread` and `ThreadPool` constructs to higher-level `Task` constructs.

SIMPLIFIER converts `Task`-based code into higher-level parallel design patterns: `Parallel.For(Each)` and `Parallel.Invoke`.

ASYNCIFIER refactors callback-based asynchronous constructs to new `async/await` keywords.

When developers manually migrate their legacy software to modern concurrency constructs, they introduce misuses. We present ASYNCFIXER, an interactive program transformation tool that detects and fixes 14 common `async/await` misuses.

(3) These program transformations are useful in practice for both industry and open-source communities.

Developers of the open-source software accepted 408 patches which are generated by our tools. We received positive feedback from early users.

Our tools are also embraced by two companies. They started to actively use AsyncFixer in their automated build process AsyncFixer detected and fixed 169 misuses in proprietary code-bases.

We now present our plans for possible future work building upon our current contributions and results as described in Chapters 3, 4, and 5:

8.2 Understanding Runtime Properties of Concurrent Code

So far we statically analyzed the source code to understand how programmers use concurrency. Although static analysis offered very interesting findings in our studies, it offered a limited insight in true intricacies of concurrency libraries' usage. The key motivation for developing concurrent applications is to deliver superior performance in comparison to sequential applications. Analyzing this fundamental question – the speedup obtained from parallel libraries – cannot be based on static source code analysis alone. To the best of our knowledge, there is no empirical study which dynamically inspects parallel library usage on a large-scale.

We plan to dynamically analyze a subset of our previously analyzed C# code repository. We only select the applications which use many concurrency constructs and have unit tests which ensure high execution coverage. Because the executions considered by a dynamic analysis is exactly those that are triggered by the test suite, the selection of the unit tests is critical. When the test suite is not rich enough, we plan to research techniques to automatically carve out performance tests from executions of an application under a realistic load. During the execution of the unit tests, .NET profilers can be used to gather the following runtime information:

1. CPU utilization: Low CPU usage could be a sign of concurrency bugs such as deadlocks and synchronization overhead. Conversely, a high CPU utilization does not necessarily indicate success, as livelocks result in high CPU utilization.

2. Garbage collection: For managed applications, too many memory management operations can hinder actual work execution and could point to faulty design or implemen-

tation.

3. Total thread execution time: This enables us to approximate the total execution time if the program was to be executed sequentially. By comparing the total thread execution time to the elapsed time of sequential implementation of the same program, we can understand the parallelization overhead.

We plan to collect the runtime information from the unit tests that are executed both in parallel and sequential. However, in most cases, we only have access to the concurrent implementations of the programs. To emulate the sequential execution, we can set `MaxDegreeOfParallelism` to 1. This limits the number of concurrent operations to 1, which enables sequential mode. However, switching to serial execution could pose other challenges. For example, it could trigger deadlocks in programs where a thread will wait indefinitely for a signal from another concurrent thread. Such deadlocks would not occur under the concurrent execution. We plan to adapt deadlock strategies (e.g., setting upper bounds on execution time, roll-backs, etc.) to recover from such cases. After collecting runtime information and comparing the performance of concurrent mode with sequential mode, we can answer many questions: are programmers getting speedup from parallel execution? Why or why not? What are some bad practices? Answering such questions can uncover very interesting facts and give clues about concurrency bugs in concurrent applications.

8.3 Runtime Implications of `async/await`

If we only rely on static analysis, we might miss many misuses of `async/await`. For instance, developers unknowingly execute long-running operations on UI thread due to new `async` paradigm. We found that developers (almost) always capture UI context in `await` statements even though they do not update UI in the rest of the method. This causes the rest of the method executes on UI thread, thus running long operations can make the UI unresponsive. The must accurate way to detect these long-running operations is performing dynamic analysis. We plan to implement a profiler that analyzes top-level event handlers in order to detect long-running operations on UI thread.

The profiler also enables us to understand whether there are many separate accesses to UI thread. This can cause sluggishness as responsiveness suffers from thousands of paper cuts. Hence, it is important to combine UI update statements to eliminate this overhead of context-switching. We can warn the developer and guide him to combine these UI accesses.

8.4 Context Decider Tool

We noticed that the source of many `async/await` misuses is the lack of thread-execution knowledge. Developers are always in doubt about whether the statements following the `await` keyword should execute on (i) the UI & ASP.NET request threads or (ii) threadpool threads. We propose to develop an analyzer which decides whether or not the developer should capture the context at the `await` point by calling the `ConfigureAwait(false)`. One way of doing this is to recursively check the call graph after the `await` statement. If the call graph does not need the UI or ASP request thread context, then developer should use `ConfigureAwait(false)` and the rest should be executed on the threadpool. This will definitely increase the performance of the software. The main question here is how the tool understands whether the code needs the synchronization context. For instance, for the UI thread, the analyzer will check the symbol of every member access in the call graph whether they are derived from `System.Windows` classes. Accessing or modifying the components from those classes potentially require the context. This approach is based on heuristics and it will allow false-negatives by being on the safe side.

This analyzer can be coupled with a visualizer which colorizes the expressions and methods based on what kind of thread executes them in the current program:

1. Methods or expressions are executed only on the UI & ASP.NET request threads (synchronization context).

2. Methods or expressions are executed only on the thread pool threads.

3. Methods or expressions can be executed on both UI/ASP or thread pool threads.

4. Undecidable ones at the compile-time.

8.5 Guiding Programmers to Parallelize the Code

According to the previously mentioned Intel survey [113], 33% of participants said that "[determining] where parallelization should occur" is the most important feature they would like to see in a tool. Programmers are most interested in tools that would guide and help them parallelize their code via an interactive approach.

We propose an *interactive* tool that exposes parallelism by identifying independent code regions in methods or loops. Such a tool helps programmers discover opportunities for parallelism in their code. Based on the opportunities, the tool suggests several refactorings for improving performance via pipelines with TPL dataflow, fork/join tasks, loop parallelism, etc.

This approach is also important for exploiting fine-grained parallelism with `async/await` constructs. Programmers can use `await` keywords too early in the methods, so that other independent code fragments cannot execute in parallel. Even though the control is passed to the caller in `await` points and maintain responsiveness, it is not exploiting fine-grained parallelism. The location where the programmers introduce `await` keyword in the `async` methods is critical for performance.

In the example code below, the method `AccessTheWebAsync` asynchronously gets the content of an url. At line 3, there is an asynchronous operation and it returns a `Task`. This task needs to be completed until line 4, where `await` keyword is used. If the task is not completed by the line 4, `await` is a blocking call and waits for the task completion. Hence, between line 3 and 4, the programmer might put some independent operations. In the code below, `DoIndependentWork` should be introduced between these lines to take advantage of fine-grained parallelism. Our approach that identifies independent code regions can determine where to put the `await` keyword for the best degree of parallelism.

```
1 async Task<int> AccessTheWebAsync() {
2   HttpClient client = new HttpClient();
3   Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");
4   string urlContents = await getStringTask;
5   DoIndependentWork(); // Independent work from GetStringAsync.
6   return urlContents.Length;
7 }
```

Our approach will be based on static source code analysis (dependency, escape analysis) enhanced with dynamic information and programmer feedback, and, in some cases, supported by code contracts. The runtime information provided by the profile-driven dynamic analysis can help with the imprecision of the static analyses used in refactoring tools. Method properties (e.g., pure methods) specified with the code contracts can also be helpful.

Performing this analysis based on static code examination is exactly the same problem that auto-parallelizing compilers face in discovering parallelizable code. These compilers have only been successful at parallelizing small and straight-forward kernel loops, but have been unsuccessful at introducing meaningful parallelism in large, irregular, non-scientific applications. Meaningful parallelism requires deep understanding of the program invariants, along with the data- and control-flow relationships between parts of the program. Using profile-driven dynamic analysis we could overcome these limitations of auto-parallelizing compilers (e.g., overly conservative), enabling us to increase program comprehension and to identify more application parallelism.

One of the main challenges here is correlating the low-level information (e.g., specific memory accesses, branch operations) gathered during program execution to the high-level data and control flow information. The other challenge is that profile-driven analysis can only provide accurate dependence information for a specific input. It is important to run the program with a representative input, such that all important code regions are exercised and the most important data dependencies can manifest. For the critical "may" data dependences, we propose an interactive approach that receives feedback from the programmer, who has domain knowledge and sees the big picture.

REFERENCES

[1] "Task parallel library (tpl)." [Online]. Available: http://msdn.microsoft.com/en-us/library/dd460717.aspx

[2] "Parallel language integrated query (plinq)." [Online]. Available: http://msdn.microsoft.com/en-us/library/dd460688.aspx/

[3] "System.Threading." [Online]. Available: http://msdn.microsoft.com/en-us/library/system.threading

[4] "Threading building block (tbb)." [Online]. Available: http://threadingbuildingblocks.org/

[5] C. Campbell, R. Johnson, A. Miller, and S. Toub, *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 2010.

[6] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2393596.2393660 pp. 54–65.

[7] W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, and F. Castor, "Are Java programmers transitioning to multicore?: a large scale study of java FLOSS," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, ser. SPLASH '11 Workshops. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2095050.2095072 pp. 123–128.

[8] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande - JAVA '00*. New York, New York, USA: ACM Press, June 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=337449.337465 pp. 36–43.

[9] D. Syme, T. Petricek, and D. Lomov, "The F# asynchronous programming model," in *Proceedings of the 13th international conference on Practical aspects of declarative languages*, ser. PADL'11, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1946313.1946334 pp. 175–189.

[10] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen, "Pause n Play: Formalizing Asynchronous CSharp," in *Proceedings of the 26th European conference on Object-Oriented Programming*, ser. ECOOP '12, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31057-7_12 pp. 233–257.

[11] "Scala async." [Online]. Available: http://docs.scala-lang.org/sips/pending/async.html

[12] "Microsoft - asynchronous programming patterns." [Online]. Available: http://msdn.microsoft.com/en-us/library/jj152938.aspx

[13] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

[14] "Best practices in asynchronous programming." [Online]. Available: http://msdn.microsoft.com/en-us/magazine/jj991977.aspx

[15] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen, "A study and toolkit for asynchronous programming in c#," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568309 pp. 1117–1127.

[16] "Github." [Online]. Available: https://github.com

[17] "Visual studio." [Online]. Available: https://beta.visualstudio.com/vs/

[18] "The roslyn project." [Online]. Available: http://msdn.microsoft.com/en-us/hh500769

[19] S. Okur, C. Erdogan, and D. Dig, "Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions," in *ECOOP '14*, ser. Lecture Notes in Computer Science, R. Jones, Ed., vol. 8586. Springer Berlin Heidelberg, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44202-9_21 pp. 515–540.

[20] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs journal*, vol. 30, no. 3, pp. 202–210, 2005.

[21] D. Lea, "The java. util. concurrent synchronizer framework," *Science of Computer Programming*, vol. 58, no. 3, pp. 293–309, 2005.

[22] "Collections.concurrent (cc)." [Online]. Available: http://msdn.microsoft.com/en-us/library/dd997305.aspx/

[23] S. Toub, "Patterns of Parallel Programming," *Microsoft Corporation*, 2010. [Online]. Available: http://download.microsoft.com/download/3/4/D/34D13993-2132-4E04-AE48-53D3150057BD/Patterns_of_Parallel_Programming_VisualBasic.pdf$\delimiter"026E30F$npapers2://publication/uuid/DC334E2E-E563-4824-8471-3F7EA232424A

[24] "Tiraggo repo." [Online]. Available: https://github.com/BrewDawg/Tiraggo

[25] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," *ACM SIGPLAN Notices*, vol. 44, no. 10, p. 227, Oct. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1639949.1640106

[26] "Passwordgenerator repo." [Online]. Available: https://passwordgenerator.codeplex.com/

[27] D. Leijen and J. Hall, "Parallel Performance: Optimize Managed Code For Multi-Core Machines," *MSDN*, Oct. 2007. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

[28] "Ravendb - the open source nosql database for .net." [Online]. Available: http://ravendb.net

[29] "Appvisum repo." [Online]. Available: https://github.com/Alxandr/AppVisum

[30] "Oracle - javaawt." [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html

[31] "Oracle - javaswing." [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/swing/

[32] "Windows forms." [Online]. Available: http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx

[33] "Windows presentation foundation." [Online]. Available: http://msdn.microsoft.com/en-us/library/ms754130.aspx

[34] "Nuget package management system." [Online]. Available: http://www.nuget.org/

[35] "Stack overflow." [Online]. Available: http://stackoverflow.com

[36] "Dynamo repo." [Online]. Available: https://github.com/ikeough/Dynamo

[37] "Antlrcs repo." [Online]. Available: http://github.com/antlr/antlrcs

[38] "Kudu repo." [Online]. Available: https://github.com/projectkudu/kudu

[39] "Lucene.net repo." [Online]. Available: https://github.com/apache/lucene.net

[40] "Jace repo." [Online]. Available: https://github.com/pieterderycke/Jace

[41] "Windows store." [Online]. Available: http://www.windowsphone.com/en-us/store

[42] "Gartner." [Online]. Available: http://www.gartner.com/newsroom/id/2153215

[43] "Codeplex." [Online]. Available: http://codeplex.com

[44] "Survival of the forgest." [Online]. Available: http://redmonk.com/sogrady/2011/06/02/blackduck-webinar/

[45] "Wpcollector source code." [Online]. Available: https://github.com/semihokur/wpcollector

[46] J. Albahari and B. Albahari, *CSharp 5.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2012.

[47] "Cimbalino-phone-toolkit repo." [Online]. Available: https://github.com/Cimbalino/Cimbalino-Phone-Toolkit

[48] "Indulged-flickr repo." [Online]. Available: https://github.com/powerytg/indulged-flickr

[49] "iracermotioncontrol repo." [Online]. Available: https://github.com/lanceseidman/iRacer_MotionControl

[50] "Adsclient repo." [Online]. Available: https://github.com/roelandmoors/adsclient

[51] "Ocell pull request." [Online]. Available: https://github.com/gjulianm/Ocell/pull/27

[52] "Phoneguitartab repo." [Online]. Available: http://phoneguitartab.codeplex.com/

[53] "Cimbalino pull request." [Online]. Available: https://github.com/Cimbalino/Cimbalino-Phone-Toolkit/pull/21

[54] "Softbuild.data repo." [Online]. Available: https://github.com/CH3COOH/Softbuild.Data

[55] "Kanboxwp repo." [Online]. Available: https://github.com/jarvisji/kanboxwp

[56] "32feet repo." [Online]. Available: http://32feet.codeplex.com

[57] "Playerframework repo." [Online]. Available: http://playerframework.codeplex.com/

[58] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, Dec. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=2036658.2036677

[59] "Our companion website." [Online]. Available: http://learnasync.net

[60] "Languagedetectapp repo." [Online]. Available: https://github.com/7ung/LanguageDetectApp

[61] "Using async for file access (c#)." [Online]. Available: https://msdn.microsoft.com/en-us/library/mt674879.aspx

[62] "Waslibs repo." [Online]. Available: https://github.com/wasteam/waslibs

[63] "native-windows-phone-sdk repo." [Online]. Available: https://github.com/eTilbudsavis/native-windows-phone-sdk

[64] "using statement (c# reference)." [Online]. Available: https://msdn.microsoft.com/en-us/library/yh598w02.aspx

[65] "Nbitcoin repo." [Online]. Available: https://github.com/MetacoSA/NBitcoin

[66] "Windows10iotadafruitled repo." [Online]. Available: https://github.com/timothystewart6/Windows10IoTAdaFruitLed

[67] "Orleankka repo." [Online]. Available: https://github.com/OrleansContrib/Orleankka

[68] "app-crm repo." [Online]. Available: https://github.com/xamarin/app-crm

[69] "Oscill repo." [Online]. Available: https://github.com/Alexx999/Oscill

[70] "Codesmithtools." [Online]. Available: http://www.codesmithtools.com/

[71] "Roslyn tail await optimization." [Online]. Available: https://github.com/dotnet/roslyn/issues/1981

[72] S. Karus and H. Gall, "A study of language usage evolution in open source software," in *Proceeding of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1985441.1985447 pp. 13–22.

[73] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of Java generics," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1047–1089, 2013. [Online]. Available: http://link.springer.com/10.1007/s10664-012-9236-6

[74] M. Hoppe and S. Hanenberg, "Do Developers Benefit from Generic Types? An Empirical Comparison of Generic and Raw Types in Java," *OOPSLA '13*, pp. 457–474, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2509528

[75] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger, "How developers use the dynamic features of programming languages," in *Proceeding of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1985441.1985448 pp. 23–32.

[76] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale Java open source code repository," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1852786.1852801 pp. 11–21.

[77] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features," in *ICSE '14*, ser. ICSE 2014. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568295 pp. 779–790.

[78] R. P. Buse, "Automatically describing program structure and behavior," Ph.D. dissertation, University of Virginia, 2012.

[79] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy, "Software engineering for multicore systems: an experience report," in *Proceedings of the 1st International Workshop on Multicore software engineering*, ser. IWMSE '08, 2008, pp. 53–60.

[80] L. Hochstein, F. Shull, and L. B. Reid, "The role of mpi in development time: a case study," in *SC Conference*, 2008, pp. 1–10.

[81] "Dotnetwebtoolkit repo." [Online]. Available: https://github.com/chrisdunelm/DotNetWebToolkit

[82] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. Online Edition, 2011.

[83] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '08, 2008, pp. 329–339.

[84] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 77–88.

[85] 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.

[86] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *The 40rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '10, 2010, pp. 221–230.

[87] T. Dey, W. Wang, J. Davidson, and M. Soffa, "Characterizing multi-threaded applications based on shared-resource contention," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, 2011, pp. 76–86.

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

[89] D. Dig, J. Marrero, and M. Ernst, "How do programs become more concurrent: a story of program transformations," in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, ser. IWMSE '11, 2011, pp. 43–50.

[90] D. Dig, J. Marrero, and M. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 397–407.

[91] Y. Lin and D. Dig, "A study and toolkit of CHECK-THEN-ACT idioms of Java concurrent collections," *Software Testing, Verification and Reliability*, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1567

[92] M. De Wael, S. Marr, and T. Van Cutsem, "Fork/Join Parallelism in the Wild: Documenting Patterns and Anti-patterns in Java Programs Using the Fork/Join Framework," in *PPPJ '14*, ser. PPPJ '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2647508.2647511 pp. 39–50.

[93] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '06, 2006, pp. 37–48.

[94] S. Park, R. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10, 2010, pp. 245–254.

[95] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[96] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, 2012, pp. 485–502.

[97] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal API usage patterns," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11, 2011, pp. 456–459.

[98] L. Wendehals and A. Orso, "Recognizing behavioral patterns atruntime using finite automata," in *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, ser. WODA '06, 2006, pp. 33–40.

[99] W. Griswold and D. Notkin, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, University of Washington, Seattle, WA, 1992. [Online]. Available: http://en.scientificcommons.org/42910995

[100] Y. Lin, C. Radoi, and D. Dig, "Retrofitting Concurrency for Android Applications Through Refactoring," in *FSE '14*, ser. FSE 2014. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635903 pp. 341–352.

[101] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Proceedings of the Foundations of Software Engineering*, ser. FSE '13, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2491411.2491461 pp. 543–553.

[102] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "Relooper," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1639950.1640018 pp. 793–794.

[103] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33–38, July 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209

[104] "Resharper." [Online]. Available: http://www.jetbrains.com/resharper/

[105] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *OOPSLA'05*, vol. 40, no. 10. New York, New York, USA: ACM Press, Oct. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1094811.1094832

[106] "Google trends." [Online]. Available: https://www.google.com/trends

[107] "The T. J. Watson libraries for analysis." [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page

[108] "Eclipse Java development tools." [Online]. Available: https://eclipse.org/jdt

[109] "a C language family frontend for LLVM." [Online]. Available: http://clang.llvm.org

[110] "Eclipse CDT." [Online]. Available: https://eclipse.org/cdt

[111] "Github for Developers." [Online]. Available: https://developer.github.com/v3

[112] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 224–235.

[113] Intel, "The Parallel Programming Landscape," *The State of Parallel Programming*, 2012.