Technical Report

UCAM-CL-TR-974
ISSN 1476-2986

Number 974

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Dynamic analysis for concurrency optimisation

Indigo J. D. Orton

August 2022

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted October 2021 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Hughes Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Dynamic Analysis for Concurrency Optimisation

Indigo Jay Dennis Orton

Modern software engineering is, broadly, a continuous activity – many pieces of industrial software are constantly developed, they are never "finished". This process of constant improvement necessitates small, incremental changes to ensure stability and maintainability of the software and its codebase. This includes incremental changes to improve performance. In this thesis I focus on improvements to the efficiency of concurrency usage, at a source-code level, within a piece of software. These improvements are challenging to identify and implement as concurrency and performance behaviour is only exhibited at runtime, thus requiring *dynamic* analysis, whilst making incremental changes requires *static* source-code patches. This challenge is compounded as a codebase evolves, as the efficiency of various concurrency uses may change – an instance of concurrency that was previously beneficial may become inefficient due to the evolution of the software. I present an automatic-program-analysis methodology to identify potential performance improvements, estimate their quantitative effect, and generate static source-code patches to implement them. Using a proof-of-concept implementation, I present evaluations that demonstrate the methodology's efficacy.

Multicore processors are the default for modern computers and leveraging this concurrency is a key aspect of modern software. Effective use of concurrency can significantly improve software performance, though the inverse is also true – ineffective use can impair software performance. However, as the saying goes "concurrency is hard"; it is fundamentally difficult to statically reason about, let alone optimise. Indeed, many of its properties, especially those related to performance, are only exhibited at runtime.

In this thesis I explore the use of *dynamic analysis for concurrency optimisation*. I argue this field is under-explored, yet represents a substantial opportunity for improving software performance. A key challenge within this field, and one that extends beyond concurrency, is the generation of static changes (e.g. source-code changes) from dynamic analysis. The gap between the static and dynamic domains is well studied in terms of using static analysis to improve dynamic analysis efficiency and using dynamic analysis to confirm static analysis hypothesises (e.g. race-condition detection), however, I argue the gap is understudied when transitioning from the dynamic to the static domain.

Acknowledgements

Thank you to Alan Mycroft, my supervisor, for being an indefatigable guide and source of encouragement throughout. In particular, I want to acknowledge that this thesis has only been possible as Alan decided to take on an excitable Australian during his transition away from full-time academia. My life has been enriched by the experience, the topic, and the supervisor. With a mind to Alan's perennially reoccurring advice to *be concise*, I'll tie this up quickly.

Thank you to my family for their support and unfailingly good humour, especially throughout these pandemic times. It is still debated just how important cryptic crosswords are to a good thesis, but we've left no space for regret.

Thank you to my friends, near and far, for the many distractions and happy procrastinations.

Dankie Christine vir alles. Lewe is beter met jou. Lewe is voller met jou.

To my parents, Barb and Tim.
For life, genes, and curiosity.

Contents

1 Introduction **13**
- 1.1 Thesis structure . . . 17
- 1.2 Thesis contextualisation . . . 18
  - 1.2.1 Thesis statement . . . 18
  - 1.2.2 Running scenario . . . 19
- 1.3 Findings . . . 21
- 1.4 Published material . . . 21

2 Technical background **23**
- 2.1 Program analysis . . . 23
  - 2.1.1 Static analysis . . . 24
  - 2.1.2 Dynamic analysis . . . 25
  - 2.1.3 Combining static and dynamic analysis . . . 26
- 2.2 Dynamic analysis for static optimisation . . . 26
- 2.3 Tracing, sampling, and dynamic data . . . 27
- 2.4 Concurrency models . . . 28
  - 2.4.1 Abstract concurrency model . . . 29
  - 2.4.2 Java 8 implementation and background . . . 32
  - 2.4.3 Related concurrency models . . . 32
- 2.5 Java and the JVM . . . 37
- 2.6 Experimental environment . . . 37
  - 2.6.1 *Acme* – Real-world evaluation target . . . 38

3 Execution Tracer **41**
- 3.1 Introduction . . . 41
  - 3.1.1 Target overhead . . . 44
- 3.2 Trace-log . . . 45
- 3.3 Sparse tracing . . . 45
- 3.4 Tracer . . . 48
  - 3.4.1 Lockless buffer exchange protocol . . . 49
    - 3.4.1.1 Algorithmic specification . . . 51
- 3.5 Evaluation . . . 55
- 3.6 Related work . . . 59
- 3.7 Conclusion . . . 62

4 Identifying Concurrency Improvements with Dynamic Analysis 63
4.1 Introduction . . . 63
4.2 Method . . . 67
4.2.1 Dynamic context . . . 68
4.2.2 Task groups . . . 68
4.2.3 Optimisations . . . 70
4.2.4 Estimation . . . 73
4.2.4.1 Soundness . . . 75
4.2.4.2 Trace-DAG construction . . . 75
4.2.4.3 Optimisation graph-edits . . . 77
4.2.4.4 Trace-DAG to Trace-log derivation . . . 79
4.2.4.5 Sleep estimation . . . 79
4.2.5 Improvement performance effect . . . 82
4.2.6 Multiple improvements . . . 84
4.2.7 Measuring and Selecting Improvements . . . 84
4.3 Evaluation . . . 85
4.3.1 Micro-benchmarks . . . 86
4.3.2 Estimation Accuracy . . . 92
4.3.3 Estimation consistency . . . 95
4.3.4 Suggested Improvements . . . 98
4.4 Related work . . . 100
4.4.1 Performance prediction . . . 100
4.4.2 Concurrency analysis . . . 101
4.5 Discussion . . . 103
4.5.1 Sleep estimation and full trace-logs . . . 103
4.5.2 Human refactoring . . . 103
4.5.3 Is this an artefact of Java's thread implementation? . . . 104
4.5.4 Could better implementation/developer practice avoid these problems? 104
4.6 Conclusion . . . 104

5 Source-Code Patches from Dynamic Analysis 107
5.1 Introduction . . . 107
5.2 Running example . . . 110
5.3 Method . . . 111
5.3.1 Abstract Program Graph . . . 112
5.3.2 Dynamic-Static Mapper . . . 121
5.3.3 Change Transformation Functions . . . 123
5.3.4 Rendering Source Code . . . 124
5.4 Real-world complexity . . . 125
5.5 Application to real-world . . . 127
5.6 Related work . . . 128
5.7 Discussion . . . 128
5.8 Conclusion . . . 131

6 Tracing Overhead and Observer Effects 133
6.1 Introduction . . . 133
6.2 Uniformity in Tracing . . . 135
6.3 Experimental Method . . . 136

6.3.1 Configurable overhead . . . 136
6.3.2 Concurrency-performance analyser . . . 136
6.4 Experimental results . . . 137
6.4.1 Configurations . . . 137
6.4.2 Metrics . . . 138
6.4.3 Experiments . . . 139
6.4.4 Limitations . . . 144
6.5 Related work . . . 144
6.6 Discussion . . . 145
6.6.1 Functional effects of overhead . . . 145
6.6.2 Practical effects of overhead on developers . . . 145
6.7 Conclusion . . . 146

7 Discussion **147**
7.1 Runtime and language generalisation . . . 147
7.2 Concurrency model generalisation . . . 148
7.3 Simplicity and optimality; cost and benefit . . . 149
7.4 Estimation is good, so long as its accurate . . . 150
7.5 Analysis at the developer's abstraction . . . 150
7.6 Reversing the pipeline . . . 151
7.7 The challenges of the real world . . . 151

8 Conclusion **153**
8.1 An exciting world . . . 154

Bibliography **155**

Chapter 1

Introduction

Concurrency is integral to modern software performance and optimisation. The decline of Moore's Law [114, 111, 34], or more specifically the decline of exponential single-core performance increases, has forced software to better utilise multicore processors. This proliferation of concurrency [34] has, naturally, caused a proliferation of concurrency-based inefficiencies – we tripped over ourselves in our excitement about the promised land of concurrent systems. These inefficiencies waste processing power and can defeat the original purpose of introducing concurrency, improving performance. In fact, in many cases, removing certain instances of concurrency can improve overall program performance (by freeing system resources for other use).

Optimising concurrency usage, to deal with these inefficiencies, is non-trivial as concurrent behaviour can be unintuitive and statically non-obvious [34] – reading the code will not necessarily provide a useful intuition of what occurs at runtime in a concurrent system, unlike sequential code which is more straightforward. Importantly, many properties of concurrent behaviour and software performance are only exhibited at runtime, making static analysis [75] (manual or automated) insufficient. For example, how long one concurrent operation takes to execute, and therefore how long another operation waits for its result, is not statically determinable (outside of trivial cases such as explicit sleep operations).

In this thesis I am interested in investigating the use of dynamic analysis [75] for statically optimising programs, for example, by generating static code patches. This is a broad topic that encompasses existing examples, such as Just-In-Time (JIT) compilation [5, 51], and many as yet undeveloped potential optimisations. Unlike JIT compilers, I am interested specifically in source-level static optimisations that can be applied during development.

However, static optimisations is a broad topic so I take a narrower perspective to explore a sub-field that can be instructive for the broader topic. Specifically, I explore the use of *dynamic analysis for concurrency optimisation*, an interesting sub-field of dynamic analysis for static optimisation. Concurrency performance optimisation is well suited to dynamic analysis given both concurrency behaviour and performance are only exhibited at runtime. I argue that this field is understudied, despite its substantial opportunity for improving software performance.

An aspect of this field, that extends beyond it to the broader topic of dynamic analysis for static optimisation, is the generation of static changes (e.g. source-code patches) from dynamic analysis. The gap between the static and dynamic domains is well studied in terms of using static analysis to improve dynamic analysis efficiency and using dynamic analysis to confirm static analysis hypothesises (such as in race-condition detection [17, 16, 64]).

I argue, though, that the gap is understudied when starting with dynamic analysis and transitioning to the static domain.

A particular example I focus on in this thesis is the use of dynamic analysis, based on execution traces, to identify and remove inefficient uses of concurrency. In effect, this optimisation aims to reduce the amount of time threads spend sleeping by removing unnecessary, and inefficient, concurrency. Sleeping threads consume resources without progressing the computation of the program (Urma et al. [121] Section 15.2.3) and can have more severe adverse effects in certain circumstances (such as deadlock via thread starvation). To investigate dynamic analysis for concurrency optimisation, and this example in particular, I undertake the *theoretical development of a practical technique* called *Dynamic-Trace Refactoring for Static Optimisation* (*DTRSO*) and develop a proof-of-concept implementation to evaluate it in the real-world.

A key idea of *DTRSO* is to *estimate* the performance effects of possible optimisations, rather than attempting to identify potentially inefficient source-code patterns. Identifying a potentially inefficient pattern is not especially useful as it is unclear whether modifying the code will improve or degrade performance. Whereas, by accurately estimating the performance effect of a change to some code, we can rationally decide whether to make the change or not. This contrasts with bug-detection analysis [9, 57, 23, 58] in which it is, sometimes, acceptable to identify *potential* bugs that should be fixed to avoid that bug ever occurring; identifying changes that could only *potentially* improve performance is not much use. Furthermore, false positives in bug-detection program analysis are significant barriers to adoption [19] and it appears likely that this effect would be similar, if not more significant, for performance program analysis (benchmarking a program before and after implementing a suggested change is a significant cost to a developer[1]).

DTRSO centres on dynamic analysis of execution traces, to accurately estimate the effect of potential optimisations (Chapter 4), and then bridging the dynamic-to-static divide to generate static source-code patches (Chapter 5). The core methodological tool is the refactoring of traces to approximate the trace that the program would generate given some change. This is done by converting the trace into a graph, which encodes constraints and costs (as time delays between events), and then modifying edges in the graph to approximate a change (see Section 4.2.4). This allows estimation of potential changes *without re-executing the program* and results in highly accurate estimation of performance. *DTRSO* has three stages (illustrated in Fig. 1.1): tracing (*Quilt* – Chapter 3), analysis (*Rehype* – Chapter 4), and patch generation (*Scopda* – Chapter 5). I have developed an implementation[2] of *DTRSO* to enable investigation of the various research questions addressed in this thesis.

This thesis makes four contributions, the three individual components of *DTRSO* and an analysis of the effect of tracing overhead on program behaviour and program analysis. The three *DTRSO* components are: the low-overhead tracer; concurrency performance analyser that refactors traces to identify improvements and accurately estimate their effects; and a tool to generate static source-code patches for dynamic-analysis-specified improvements. The fourth contribution is an analysis of tracing overhead, its natural observer effect, and how this affects the traces collected and analysis performed using

[1] Consider the cost of executing a long-running program, multiple times, before and after making a change to assess the change's performance effect(s). Then consider the cost of doing that for each change in an arbitrarily large tree of possible changes.

[2] The implementation is approx. 65 000 lines of Rust code.

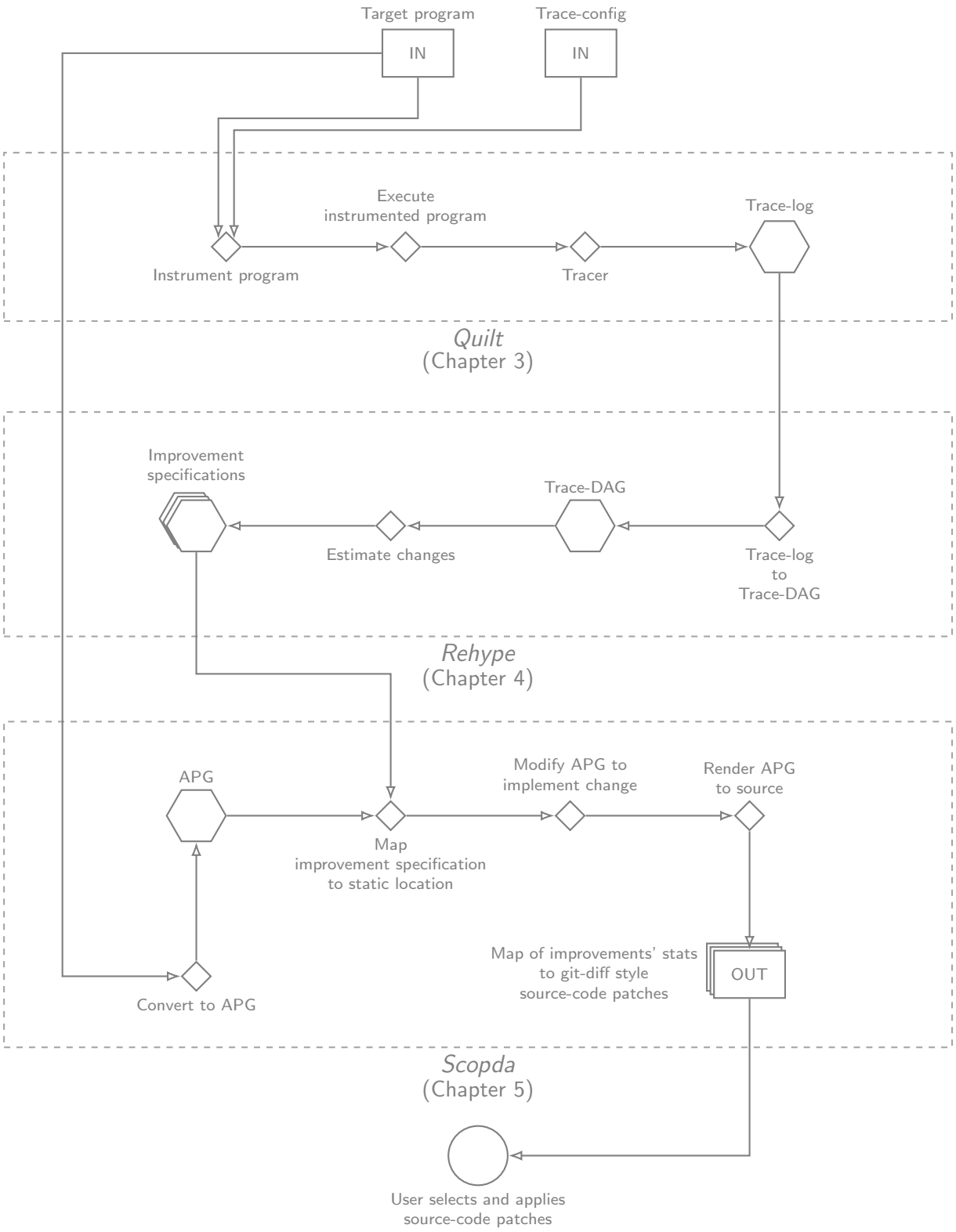


Figure 1.1: A high-level overview of the three stages of *DTRSO*: *Quilt*, *Rehype*, and *Scopda*. This diagram is reproduced in each of the three chapters presenting stages. The *Trace-DAG* (Trace Directed Acyclic Graph) is a core data structure of *Rehype* (Chapter 4). The *APG* (Abstract Program Graph) is the core data structure of *Scopda* (Chapter 5).

those traces.

Concurrency model

DTRSO focuses on task-based concurrency using thread pools and the removal of *wait-limited tasks*. Task-based concurrency is a broadly used concurrency pattern and is supported by many languages (an increasing number of languages natively support it using the `async/await` terminology). There are three key components in this concurrency model:

Tasks are units of work that may return some value. These can be thought of as a pair of a function and a set of values to be passed as arguments to the function.

Threads are logical workers that can execute functions. We will specifically deal with logical, and expensive to create, threads (e.g. Java's thread model), as opposed to hardware threads or lightweight thread-style tools such as co-routines.

Thread pools are utilities that schedule tasks to be executed by a thread. Thread pools manage a collection of threads that are reused to execute multiple tasks (as opposed to generating a new thread for every task, as threads are expensive to create). In most cases, thread pools have a limited number of threads.

This model is described, along with related concurrency models, at more length in Section 2.4.

Source-level changes and correctness

DTRSO does not automatically apply optimisations, instead it generates source-code patches that a developer can trivially apply (e.g. using `git apply` [14]). Small source-code changes can have a significant performance impact but present a greater potential risk if applied without knowledge of the broader system. *DTRSO* generates source-code changes to enable larger performance improvements, but this means that it cannot provide strong guarantees of correctness as guaranteeing the effect of such changes to concurrency is very challenging, or impossible. This is in contrast to many program optimisations, such as those in optimising compilers [1], that provide strong guarantees of correctness (that the program will generate the same result with or without the optimisation) but in doing so constrain the scope of their improvements. *DTRSO* assumes that a developer will consider the correctness of any source-code changes, before incorporating them, just as they would for changes they wrote. It is worth noting that all suggested changes are sound given the program adheres to the assumed concurrency model of purely task-based coordination (set out in Section 2.4). Though, many real-world programs will only loosely adhere to this model – for example, many real-world programs will use shared resources for logging, or other small ways – and hence a developer check is important.

Further to correctness guarantees, as *DTRSO* uses execution traces to identify improvements, it is also important that the traces are representative of normal executions. This requires inputs and an execution environment that are representative of normal usage of the program. Without a representative trace the suggested changes may not, in fact, improve the program performance, regardless of their behavioural correctness. The effect of (un)representative traces is considered in Chapter 4.

```
int example(ExecutorService executorService) {
  Future<Integer> result = executorService.submit(() -> {
    // Some work.
  });
  int other; // Initialised by <other work>
  /* <other work> */
  return result.get() + other;
}
```

Figure 1.2: Simplified Java example of a potentially inefficient concurrency pattern (in Java 8 `ExecutorService.submit` is the standard way of running some code in the background). If `<other work>` takes a long time, then the use of a background task makes sense as it is parallelising work. However, if it does not take a long time, then the program may end up using two threads where one would suffice (one thread doing the work and one thread waiting for the other to finish).

Concurrency interconnectedness

Small changes to concurrency, such as inlining a task (executing a task sequentially instead of submitting it to a thread pool), can have significant effects on performance. As concurrent operations naturally interact with the rest of the system, modifying or removing them can have significant knock-on effects on other concurrent components. This is one of the key challenges in accurately estimating the effect of a change to concurrency.

Motivating concurrency example

Wait-limited tasks represent an inefficient use of concurrency within the target task-based concurrency model (more detail in Section 2.4). They are tasks that spend the majority of their execution waiting for other tasks' results. In the simple case this might be a task that immediately spawns another task and waits on it. Wait-limited tasks are inefficient [121] because they consume a thread (and its associated resources) without progressing the program's computation[3]. This is especially problematic in thread pools with limited numbers of threads as the thread pool can reach saturation (all threads being used), either throttling performance (both throughput and potentially execution speed) or, potentially, causing a deadlock.

For example, Fig. 1.2 provides an example of Java code that may, or may not, be an inefficient use of concurrency, depending on how long the `<other work>` block takes to execute. If it takes a long time, then the use of concurrency may be very beneficial by parallelising long-lived work. However, if it does not take a long time, then the program may end up using two threads where one would do.

1.1 Thesis structure

This thesis is structured around four core research chapters (Chapters 3, 4, 5, and 6).

[3]Other concurrency models, such as Fork-Join, address aspects of this problem by having threads execute other tasks while awaiting a task's result. However, these models are not always desirable, as discussed in Section 2.4.

Chapter 2 provides general technical background, such as a description of the concurrency model investigated in this thesis, and describes a program, *Acme*, used for evaluation in later chapters.

Chapter 3 describes the *Quilt* execution tracer that acts as the foundation of *DTRSO*. *Quilt* generates a *trace-log* which is then used by *Rehype* to identify improvements (as shown in Fig. 1.1).

Chapter 4 describes *Rehype*, the core concurrency-performance analyser. *Rehype* takes in *trace-logs*, estimates the effects of potential optimisations to identify improvements, and produces *improvement specifications* that are used by *Scopda* (see Fig. 1.1).

Chapter 5 describes *Scopda*, a set of methods and a tool for generating static source-code patches (git-diff style patches) from *improvement specifications* generated by *Rehype*.

Chapter 6 investigates the observer effect inherent in tracing overhead. First describing the theoretical aspects of this overhead, such as non-uniformity, and then by performing experiments with *Quilt* and *Rehype* to demonstrate the effects in the real-world.

Chapter 7 discusses points of interest, theoretical nuances that do not fit within the research chapters, potential uses of the research, and areas for further investigation.

Chapter 8 concludes the thesis by summarising and highlighting the key ideas.

Chapters 3, 4, and 5 describe the components that form my implementation of *DTRSO*. In doing so, they also each address research questions specific to their respective stages. For example, Chapter 4 considers the ability to accurately estimate performance effects of concurrency changes and the limitations of execution trace refactoring, while Chapter 5 addresses bridging the dynamic-to-static gap, a gap regularly bridged in the other direction (discussed in Chapter 2).

1.2 Thesis contextualisation

This thesis aims to address a fundamentally practical problem, how to improve software performance by improving concurrency usage, and in doing so tackles a number of theoretical challenges, such as bridging the dynamic-to-static gap (Chapter 5). This problem, and decisions made in this thesis, is best understood within the context of the thesis statement (Section 1.2.1). I give a bit more colour and intuition to this statement via an imaginary scenario (Section 1.2.2) which aims to provide a concrete setting for decisions throughout the thesis.

1.2.1 Thesis statement

The rationale of much of the work within this thesis can be captured in the thesis statement from the beginning of the abstract. This statement is referenced throughout the thesis. For simplicity, I reproduce it here:

Modern software engineering is, broadly, a continuous activity – many pieces of industrial software are constantly developed, they are never "finished". This process of constant improvement necessitates small, incremental changes to ensure stability and maintainability of the software and its codebase. This includes incremental changes to improve performance. In this thesis I focus on improvements to the efficiency of concurrency usage, at a source-code level, within a piece of software. These improvements are challenging to identify and implement as concurrency and performance behaviour is only exhibited at runtime, thus requiring *dynamic* analysis, whilst making incremental changes requires *static* source-code patches. This challenge is compounded as a codebase evolves, as the efficiency of various concurrency uses may change – an instance of concurrency that was previously beneficial may become inefficient due to the evolution of the software. I present an automatic-program-analysis methodology to identify potential performance improvements, estimate their quantitative effect, and generate static source-code patches to implement them. Using a proof-of-concept implementation, I present evaluations that demonstrate the methodology's efficacy.

1.2.2 Running scenario

To provide contextualisation and intuition for various decisions made throughout this thesis and its overall motivation, I describe an imaginary industrial scenario below. The narrative of the scenario is updated in each relevant chapter to position the chapter's content within the intended industrial context.

Our imaginary scenario centres on a software engineer, called Banjo, working on a large piece of industrial software, called Paterson. Banjo works at a large technology company, with hundreds of engineers, which develops Paterson. Paterson is complex, comprised of tens of millions of lines of code, and runs across multiple servers in the cloud.

We begin with Banjo being assigned to improve Paterson's performance, particularly the throughput of its web-request handling. Web server logs indicate that the speed of request handling by a server degrades as more simultaneous requests are made to the server. However, initial performance profiling indicates that the servers still have more compute and memory capacity available when this degradation begins.

Banjo knows that the web-request handling component uses task-based concurrency extensively and, furthermore, that this can be inefficient in some circumstances. Importantly, this could, Banjo hypothesises, lead to an artificial resource saturation where all threads in Paterson's thread pool are being used, limiting compute capacity, even though the hardware has not been fully saturated. Banjo performs more profiling to record the state of the thread pool across time. This profiling shows that the thread pool is, indeed, becoming fully saturated (i.e. there are no threads available for new tasks) just before performance begins to degrade.

Now Banjo has a hypothesis, and some initial evidence, that thread pool saturation is limiting throughput. However, it seems odd to Banjo that, despite the thread pool being saturated, the CPU is not 100% utilised. Because the number of threads in the pool is greater than the number of cores, N_c, the CPU should be fully utilised if at least N_c threads are active. Given the CPU is not fully utilised, the number of active threads at any point in time must, therefore, be less than N_c. The remaining occupied threads, at that point in time, must be asleep.

The key question for Banjo, then, is how to stop the thread pool from saturating before

hardware resources are saturated (CPU and memory, primarily)? There are three general options:

1. increase the thread pool size until the active threads saturate the hardware before all threads are occupied;

2. reduce the number of occupied threads that are sleeping;

3. or change the underlying thread pool implementation to mitigate the impact of thread saturation (e.g. using a fork-join thread pool, see Chapter 2).

The first option would be a short-term solution, though, it has negatives as well, such as increased memory usage, resource contention, and context switching. However, the inefficient thread usage will continue to be an issue, assuming the distribution of active threads to sleeping threads remains consistent, as more threads will be used for the tasks that spend time sleeping. The second option has the dual benefits of providing a longer term solution and being possible with localised incremental changes, but it requires more effort than increasing the thread pool size. The third option can be good in instances where it is possible (e.g. small projects), though modifying the semantics of a core structure within a large and/or complex codebase is risky and can have unintended consequences (such as introducing new bugs). As noted in the thesis statement, complex industrial software requires small incremental changes[4], and so this third option is not viable for Banjo[5].

Weighing these options, Banjo concludes that the safest route is to reduce the number of occupied threads that are sleeping, thus reducing the rate of thread pool saturation. Banjo concludes that this will only require small incremental changes that can be easily reviewed and approved by their colleagues (an essential attribute). Furthermore, each change's effects can be easily measured and are constrained to their local area within the software.

To summarise, Banjo is trying to improve the performance of Paterson, a large complex codebase that uses task-based concurrency. Banjo has identified thread pool saturation as a potential area for improvement and has decided to reduce the number of threads spending unnecessary time sleeping, to reduce the saturation rate on the thread pool, without affecting the speed of the software more generally. This is a particularly attractive approach as it can be performed in small incremental changes without introducing significant risk or disruption to the broader group of software engineers working on Paterson.

The key challenge for Banjo is identifying which parts of the source-code to modify and how. This thesis presents an approach to automatically identifying and quantifying potential changes, and generating the relevant source-code patches.

The *narrative instalments* included at the start of each contribution chapter chronicle Banjo's requirements and use of *DTRSO* at each stage of their work on improving Paterson's performance.

[4]There may be instances where incremental is not possible, but those are the exceptions.

[5] To further illustrate this point, imagine if you asked an engineer to try and improve the performance of some software and they responded with the suggestion of changing the semantics of a core component. Such a suggestion would, in most cases, be a change of inappropriate magnitude for the problem being addressed.

1.3 Findings

Using *DTRSO*, developed across Chapters 3, 4, and 5, I demonstrate that:

1. Removing certain instances of concurrency can improve overall system performance. Many such uses of concurrency may have originally been positive (improving performance when first implemented) but, due to subsequent changes, become inefficient.

2. Such performance inefficiencies can be consistently and automatically identified.

3. It is possible to accurately estimate the performance effect of source-code changes in a concurrent system.

4. Static source-code patches can be generated from dynamic analysis data; additionally cases of source-code patch ambiguity, where it is unclear how to implement a change, can be systematically identified and rectified.

5. Tracing that introduces too much overhead can disrupt concurrent behaviour and, as a result, invalidate subsequent analyses of the trace. Though, without tracing sufficient aspects of a program it is impossible to effectively identify inefficiencies.

Practically, I also use the *DTRSO* implementation and apply it to a (large) real-world, industrial server program. Implementing the suggested improvements doubles the theoretical throughput of the server API, by improving its concurrency efficiency.

These findings are early steps within the field of Dynamic Analysis for Concurrency Optimisation. There are many, likely more interesting, ideas to be developed and discoveries to be made in this field. I hope that the community will take a keener interest in this field, and the broader field of Dynamic Analysis for Static Optimisation, in the future and develop tools for improving software in new and better ways.

1.4 Published material

Aspects of this thesis have been published at peer-reviewed venues in three papers:

- Parts of Chapter 3 and the key ideas of Chapter 4 were presented at FTfJP'21 (Formal Techniques for Java-like Programs) [81].

- The key ideas underlying Chapter 5 were also presented at FTfJP'21 [82].

- The key ideas of Chapter 6 (along with relevant aspects of Chapter 3) were presented at MPLR'21 (Managed Programming Languages & Runtimes) [80].

Chapter 2

Technical background

This chapter aims to provide a general technical basis for the rest of this thesis and a reference point for certain components that are reused throughout the thesis (such as the target evaluation program, *Acme*). We discuss the general field of program analysis [75] in Section 2.1, the differences between static and dynamic analysis, and the challenges of combining them. In Section 2.2 we delve deeper into dynamic analysis for static optimisation, and then more specifically into dynamic analysis for concurrency optimisation. We review tracing, sampling, and execution control techniques for capturing dynamic data in Section 2.3. I describe the concurrency model that *DTRSO* (Dynamic-Trace Refactoring for Static Optimisation) focuses on and review some related models (Section 2.4). Though there are many potential concurrency models, and optimising each of them would be useful, I focus on a single model (task-based concurrency using thread-pool scheduling) in this thesis. In Section 2.5 we briefly review Java and the Java Virtual Machine (JVM) as the *DTRSO* implementation I develop in this thesis is designed for Java programs. Finally, I give a quick overview of *Acme* in Section 2.6.1, an industrial server program used for evaluation in later chapters.

2.1 Program analysis

There are two broad categories of program analysis, static analysis and dynamic analysis [75]. Static analysis uses the static program information (e.g. the program code), while dynamic analysis uses information regarding the operation of the program (e.g. execution traces). Most methods that work with source code can technically be classed as static analysis, even if they do not perform extensive analysis. As such, all compilers are essentially forms of static analysis, though compilers differ in the sophistication of their analyses. Whereas, dynamic analysis is less pervasive as it is a post-hoc (that is, it operates given some execution of the program) and thus there are greater barriers to usage.

In this thesis I use the term "analysis" to refer to both analysis and transformation

Static analysis is a highly developed field used by every compiler [1, 22]. Dynamic analysis, on the other hand, is less developed and used in specialised cases. Dynamic analysis is useful as it has different properties to static analysis, in particular, it can use precise information regarding an execution of a *particular* program with a *particular* set of inputs, whereas static analysis works with complete, but imprecise, information.

2.1.1 Static analysis

Static analysis operates on static program representations, such as the program source code. It is a highly developed area of program analysis and is used in every compiler. Technically, any algorithm or method using static program information could be considered static analysis – it is a broad term. Static analysis is used to prevent bugs at compile-time (e.g. with type-checking [88]), improve performance with optimising compilers [1], and warn against bad code patterns [48], among many other uses. It is particularly effective as, in many cases, it can be applied with little overhead in terms of developer effort as it runs directly on code.

Examples of static analysis include:

Data-flow analysis used to determine the different sets of calculated values and their usages (see Section 1.3 of Nielson et al. [75]). This can be useful when checking program correctness (e.g. during type checking) and optimising programs.

Control-flow analysis used to determine the possible routes through a program based on the control-flow statements, such as `if/else` statements (see Section 5.1 of Nielson et al. [75]). This can be used to determine unreachable code fragments, among many other uses.

Class-hierarchy analysis used to determine the possible classes of a value (see Snelting and Tip [104]). This is important when analysing object-oriented programs to determine which implementation(s) of a function will be called from a particular invocation.

Type checking is used to prevent bugs by ensuring that values adhere to the defined types (see Pierce [88]).

While examples of static transformations include:

Function inlining is a performance optimisation that inlines the code from a function in cases where the overhead of invoking another function is more than the overhead of inlining its contents at every call-site (see Chapter 12 of Aho et al. [1]).

Expression simplification is another performance optimisation that simplifies expressions that include redundancies (see Chapter 9 of Aho et al. [1]). For example, a variable that has a constant value (which can be identified using data-flow analysis) can be replaced with the constant value itself, removing the need to load the variable's value when performing the expression.

As noted previously, static analysis has complete, but imprecise, information about the program it is analysing. This means that it can operate based on all possible executions and edge cases in a program, but cannot determine whether they will occur or not (without actually, or effectively, executing the program). The root issue is that many problems are undecidable given complete but imprecise information, such as the halting problem, which limits the extent to which static analysis can be applied. However, the use of complete information does allow static analysis to perform guarantee-based optimisations, where the analyser guarantees that the program behaviour will not be affected by a given optimisation. Such guarantees are essential to optimising compilers that perform optimisations without checking with the developer.

2.1.2 Dynamic analysis

Dynamic analysis operates on program execution information, such as execution traces. It is less developed compared to static analysis as, I argue, it has a greater barrier to use and development in that it requires executing a program (in some form) before analysing the result. Furthermore, the types of problems it is suitable for is more restricted given its incomplete, but precise, information. Dynamic analysis is used to detect security issues [123], concurrency bugs [17], and optimise performance [105], among other uses. It is particularly well suited to concurrency analysis and performance analysis as both are exhibited at runtime and such analysis requires precise information.

One of the most common usages of dynamic analysis is Just-In-Time optimising compilers [5, 51] (referred to simply as JIT). In a simple form, a managed runtime, such as the JVM [59], will contain a JIT compiler will that uses dynamic analysis to identify "hot" code paths (those invoked frequently) and will compile pieces of non-native code to native code. The non-native code may be interpreted source code in some cases or, in many cases, a bytecode that is executed by the runtime. Compiling to native code provides a significant performance boost, but is not desirable for all code paths. It may be impossible to compile certain code paths in certain dynamically typed languages (e.g. Python [122]) to native code as value types may vary across different invocations of the code paths. Moreover, many code paths will not be executed enough to warrant compilation to native code (that is, compiling to native code would hinder performance more than having the native version would help). Hence the use of dynamic analysis to determine which code paths should be compiled.

Another form of dynamic analysis is profile-guided optimisation [63, 87], whereby performance profiles are gathered for a program by executing it numerous times. These profiles are then used by a compiler optimisation when determining how to optimise different components of the program. Specifically, the profiles provide execution frequency information for the different components (e.g. different functions) allowing the optimiser to use more precise information during optimisation, rather than heuristics.

In contrast to static analysis, dynamic analysis has precise, but incomplete, information. That is, for the information is uses, the information is precise as it relates to an exact execution of the program. However, it is incomplete as it represents only the execution that occurred, not all possible executions. This makes it effective for certain types of analysis, such as, again, concurrency and performance analysis where precise information is essential and complete information is not always necessary or helpful. However, this property also introduces a key constraint of all dynamic analyses, they are only as good as the data they use is representative. If the data is captured for an execution that is not representative of normal executions (e.g. because it is based on unusual input to the program), then the analysis may not be useful for the program in the general case. Furthermore, the use of incomplete informations means that dynamic analysis, unlike static analysis, cannot directly provide guarantee-based optimisations. Instead, it must either be augmented with static analysis to provide the guarantee, or use a separate safety mechanism, such as using the developer as an "oracle" or inserting runtime checks (e.g. as used by some JIT compilers to ensure assumptions made during optimisation hold at runtime).

2.1.3 Combining static and dynamic analysis

The combination of static and dynamic analysis is less common and presents its own interesting challenges. Some program analysers, such as JIT compilers, may *use* both static and dynamic analysis without *combining* static and dynamic analysis. For example, a JIT may use dynamic analysis to identify which interpreted functions to compile to native code and then use static analysis as part of that compilation process. However, such use of both types of analysis does not necessarily involve combining them.

Combining static and dynamic analysis involves using both in a discrete task, or using the output of one as a direct and essential input to the other. For example, a common approach in concurrency race-condition (a form of concurrency bug) detection analysis [17, 16, 64] is to identify potential race-conditions using static analysis (e.g. based on lock overlaps) and then verify them using dynamic analysis (e.g. by strategically slowing the program during execution to trigger the race-condition). The process of combining both types of analysis presents its own challenges that are related to, but also somewhat independent of, each form of analysis. In a general sense, these challenges relate to the translation between the static and dynamic domains.

These challenges are, in some cases, straightforward when going from the static domain to the dynamic domain. For example, controlling execution based on some static analysis [57] is a fairly direct translation. However, while some translations from the dynamic domain to the static domain are straightforward (such as profile-guided optimisation, discussed above), other translations from dynamic to static are significantly more challenging, as we shall see in Chapter 5.

2.2 Dynamic analysis for static optimisation

This thesis explores the field of Dynamic Analysis for Concurrency Optimisation, a sub-field of Dynamic Analysis for Static Optimisation. While JIT and profile-guided optimisation are both closely related, I argue they slightly orthogonal to Dynamic Analysis for Static Optimisation, as I define it. Specifically, they use dynamic analysis in a fairly straightforward manner to assist static optimisers by providing execution frequency information. Whereas, I argue a key attribute of methods within this field is the use of dynamic analysis as the root of the optimisation and static analysis as a tool for implementing an optimisation. In this way the methods are *dynamic-driven*, rather than *dynamic-assisted*. This distinction is important only insofar as it delineates the challenges faced in the field. I argue a primary challenge for methods in this field is bridging the dynamic-to-static domain gap, whereas in JIT and profile-guided optimisation methods the static analysis uses the dynamic data without significant challenge.

I argue that concurrency optimisation is an understudied sub-field, but that it is a particularly suitable target for dynamic analysis as concurrency behaviour and performance are both only exhibited at runtime and precise information is important for effective optimisation. In this thesis I specifically focus on optimisation via reduction of concurrency, which raises the questions "why do people overuse concurrency?" and "is this common?" The simple answer is: it is rarely obvious that concurrency is overused until dynamic analysis is performed. Even then it may be that overuse of concurrency does not impede performance until significant load is applied to the system (when system resources will be pushed to their limits). Furthermore, it may be the case that when certain instances

of concurrency were first added they were beneficial but, through the evolution of the software, they have since become impediments to performance.

2.3 Tracing, sampling, and dynamic data

Dynamic analysis relies on data captured during the execution of a program. There are three high-level techniques for capturing dynamic data: sampling (sometimes referred to as profiling, though profiling is a broader concept), tracing, and execution control. In this thesis we focus primarily on tracing (*Quilt* in Chapter 3 is a tracing method). We briefly review all three techniques here.

Sampling (aka profiling) is, arguably, the most commonly used method for analysing the performance of a program. Sampling based tools capture frequent snapshots of information about a program's execution. For example, the popular Linux `perf` tool [60] stops a process some number of times per second and snapshots data such as current call stack on threads, memory usage, and hardware counters (e.g. CPU instruction counts). The rate at which a sampler snapshots information is called its frequency and is measured in hertz (e.g. a frequency of 1KHz means capturing 1,000 snapshots per second, or 1 snapshot every millisecond). The rate of sampling is a key driver of overhead introduced by a sampler – the less a sampler stops a program, the less overhead it will introduce. As such, many samplers' frequency can be configured, enabling them to be used in a variety of environments that might otherwise be untenable (e.g. running a sampler in a production environment can be acceptable if the sampling rate is low enough as to keep overhead below some threshold). Samplers are useful as a general tool as they are easy to use, straightforward to interpret, broadly available, and have good support from other tools (e.g. Flame Graphs [30] are a popular visualisation for numerous samplers). For further reading, Oaks [78] provides an introduction to performance analysis and profiling in Java (particularly Chapter 3), while Hofer et al. [38] and Nisbet et al. [76] provide further discussion on sampling specifically within Java applications from a research perspective.

Tracing involves recording events every time they happen. For example, recording an event every time a lock is acquired, a network request is made, or an exception is thrown. Which events are of interest, and thus captured, is determined by what analysis is to be performed on the trace. The tracer in this thesis, *Quilt*, captures concurrency related events and function entry and exit events. Whereas, Castor [66] captures non-deterministic events (such as disk operations) to enable Record & Replay, Hofer et al.[39] record lock-based events to enable lock contention analysis, and Lengauer et al. [56] record memory-related data to enable analysis of memory performance issues. Where sampling aims to capture snapshots of a program to provide indicators useful for performance analysis, tracing aims to capture all occurrences of certain events (almost always a more limited set of events than sampling). Both techniques are valuable in different circumstances.

Execution control involves taking control of a program and managing the execution of the program to enable deeper analysis (e.g. debugging). This is a significantly more heavyweight technique (i.e. its overhead is generally far more substantial than sampling and tracing), but targets very different use cases. For example, Valgrind [74] enables

analysis of a program's memory usage by tracking all memory allocated, freed, and used. This enables tools built on Valgrind to automatically identify bugs, but it involves a significant overhead cost (between 400% and 10,000% overhead [74]). LLDB [62], on the other hand, takes control of a program's execution to enable manual debugging (e.g. by stopping execution at defined breakpoints).

In summary, the three techniques can be thought of broadly on a continuous spectrum, with sampling at one end, execution control at the other, and tracing somewhere in the middle. Sampling provides low-cost indicators as to what a program is doing, though, it is limited by only providing snapshots. Tracing has a higher cost in terms of overhead (as a general rule, though not always) but provides full information about the events it captures (e.g. if tracing lock acquisition, a user or analyser can be confident that all lock acquisitions are recorded, whereas with sampling no such confidence is possible). Execution control tools have a much higher overhead, but enable online control of what a program does – where tracing provides a record of what happened, these tools enable analysis as "it happens".

2.4 Concurrency models

Concurrency is the decomposition of a program into multiple components that may be executed in an arbitrary order [53]. Concurrency and parallelism are commonly substituted as synonyms, however, there is an important distinction: concurrency is concerned with how a program is logically decomposed into multiple parts, whilst parallelism is concerned with *executing* multiple operations at one time. Naturally, these concepts are closely connected as concurrent components may be executed in parallel (and, indeed, this is the practical aim of concurrency in many programming languages and runtimes). In this thesis we are interested specifically in concurrency, not parallelism, though we assume the concurrent programs being considered are executed with some level of parallelism[1].

There are multiple models of concurrency – i.e. multiple ways to decompose a program into components. A key high-level delineation of concurrency models is whether they perform *communication by synchronisation* or *synchronisation by communication*. Task-based concurrency is an example of the former approach, synchronisation occurs when a task waits on the result of another task and, in a pure form of task-based concurrency, this is the only avenue for concurrent operations to communicate. Another example of communication by synchronisation is the use of shared global variables to communicate; two threads may perform concurrent operations and update a shared global variable (protected by a guard (e.g. a mutex)), upon which they synchronise, to communicate. A good example of synchronisation by communication is the actor model [35] in which a set of concurrent "actors" independently perform computations and then asynchronously communicate by passing messages to each other. In this model, actors are not allowed to synchronise on shared global variables and instead must interact only by message passing. This removes the need for shared locks, which can be a source of inefficiency in concurrent processing.

Though there are many forms of concurrency, in this thesis we will focus on a particular

[1]This is important as it affects assumptions we make about concurrent programs (e.g. that tasks executing in parallel affect each other by consuming the same resources). Whereas, if we were concerned with concurrent program that executed without parallelism, we would presumably be less interested in shared resources and more interested in time sharing on the single execution thread (i.e. task scheduling).

model, task-based concurrency. In this model of concurrency, a program is decomposed into a set of tasks that take some input and produce some output. Tasks may, themselves, generate a set of sub-tasks during their execution. Task-based concurrency does not define how tasks should be executed. Though, in this thesis we focus on a model whereby tasks are submitted to a thread pool which then executes tasks on a thread from its managed "pool" of threads.

In this section, I first describe the abstract concurrency model we assume (i.e. that programs must adhere to) for the rest of the thesis and provide requisite background on Java 8's concurrency utilities that enable this model. I then discuss other similar and relevant concurrency models, specifically Fork-Join and Coroutines, and how they relate to this thesis.

2.4.1 Abstract concurrency model

The key restriction of this thesis's target concurrency model is that a program's concurrent communication must be solely based on *tasks* (i.e. no additional shared-memory-based communication). Non-functional operations such as logging (i.e. operations that do not affect the result of the program) are exempt from this restriction.

Tasks

Tasks are concurrent constructs that represent a function to be executed and store the result from the function when it completes. When some code attempts to access the result of a *task* it will sleep until the *task* has completed (i.e. the *task* has computed its result), thus performing synchronisation within task-based concurrency. A *task* within the target concurrency model conforms to the following restrictions:

- A *task* is "practically functional", that is, it is given all arguments required to execute the function and returns a single result, which is unaffected by which thread executes it or precisely when it is executed.
- While a *task* may have side effects (e.g. caching results or general logging), its side effects are expected to be irrelevant to the core functional result of the *task*.
- Similarly, *tasks* do not rely on/communicate via mutable global variables.
- A *task* may recursively create other *tasks*.
- A *task* may access, and wait for, the results of other *tasks*.
- A *task* is not functionally affected by execution context such as which thread it executes on, or how it is executed.
- A *task* is considered to be *constructed* when it is given to a *thread pool* to execute.

There are five operations in a task's life-cycle[2] (the full set of life-cycle operators is detailed in Section 3.2):

Task Constructed – occurs when the *task* is first constructed.

[2]For the purpose of this work we assume tasks are not cancelled.

Task Execution began – occurs when a *thread* begins executing the body of the *task*.

Task Result set/execution finished – occurs when the execution of the *task* finishes and the *task* result is set.

Task Result get began – occurs at the start of the access to the result of the *task*. If the *task* has not finished executing when another thread access its result, the accessing *thread* will sleep until the result is ready.

Task Result get finished – occurs at the end of the access to the result of the *task*.

Every task is assigned a unique *task-id* at construction. Some trace data reference task-ids to associate some types of events with specific tasks.

Thread pools

Thread pools contain a fixed number of worker *threads* and a queue of *tasks* to execute. Thread pools reuse threads to execute multiple tasks (it is much cheaper to coordinate threads than it is to create a new thread for each task). We assume that all *threads* in a *thread pool* are solely used for executing *tasks*. *Threads* exist in one of three states:

Active: the *thread* is awake and executing a task.

Waiting: the *thread* is sleeping because it is executing a *task* which is waiting for the result of some other *task* (see *result get began* above).

Unoccupied: the *thread* is sleeping while waiting to be assigned a new *task*.

Threads tell their containing *thread pool* when they can execute a new *task* (transition to the *unoccupied* state). The *thread pool* assigns new *tasks* to available *threads*. As there is a limited pool of *threads*, *threads* are reused to execute tasks.

To maximise throughput programs should have most *threads active* most of the time. While *waiting* a *thread* is consuming system resources while not actively progressing its current *task*. While *unoccupied* the *thread* may consume some system resources (e.g. memory while waiting for a task to execute), but is available to be used. When a thread finishes executing a task it transitions to the *unoccupied* state, even if another task is immediately available to execute (in this case the thread may immediately transition back to *active* if it is assigned to execute the task).

In this thesis *threads* refer to the source-code-level logical threads. Each thread is assigned a unique *thread-id* that trace data reference to connect events to the thread they occurred on. Although threads may be executed on different physical CPU cores, depending on the operating system scheduler, this is below the level of detail we are concerned with in this thesis.

Combinators

Within the subject of concurrency[3], a combinator is a declarative construct or operation that orchestrates tasks. For example, tasks A and B could be chained by passing the

[3]The term "combinator" was originally termed for "SKI combinator calculus" [103] and was subsequently adopted for use in concurrency. In this thesis we are only concerned with the concurrency version.

```
// Some processing functions.
Future<Step1Result> step1();
Future<Step2Result> step2(Step1Result param);

// The orchestration function.
Future<Step2Result> twoStepProcess() {
  return asyncTask(() -> {
    Step1Result s1Result = step1().waitForResult();
    Step2Result s2Result = step2(s1Result).waitForResult();
    return s2Result;
  });
}
```

(a) Java-esque pseudo-code that orchestrates two tasks using waits.

```
// step1() and step2() ...
// The orchestration function.
Future<Step2Result> twoStepProcessCombinator() {
  return Combinator
      .chainResultFrom(() -> step1())
      .into(s1Result -> step2(s1Result));
}
```

(b) A combinator version of the code. This version requires one less thread as it does not need one to orchestrate the tasks.

Figure 2.1: An example of task orchestration using wait-based programming and combinators.

output of task A as a parameter to task B. Task B would only be scheduled to execute after task A completes. The benefit of combinators is that they enable this orchestration to be declared before task execution, rather than requiring other orchestration code. Theoretically, perfect use of combinators would result in wait-free concurrency. However, one reason combinators are not ubiquitous is that they are less intuitive and ergonomic than the more straightforward orchestration code alternative [121] (this is discussed further in Section 7.3).

As an example, Fig. 2.1a contains Java-esque pseudo-code that uses imperative orchestration, while Fig. 2.1b shows the combinator version of the code. In the non-combinator version, the function `twoStepProcess` requires an extra thread to orchestrate the two tasks from `step1()` and `step2()`, which already use their own threads. The code examples have the same result, except the combinator version does not require the orchestration thread, so the whole process uses only one thread.

These examples demonstrate a key conceptual and performance disconnect in task-based concurrency: while the operation `waitForResult()` (named `.get()` in many implementations) is central to the conceptual use of futures (and thus tasks), it is the source of sleep-blocked threads in thread pools (i.e. *waiting* threads). To optimise performance, thread-pool-based code should never invoke `waitForResult()` directly, unless it is highly likely that the waited-for task is complete (so `waitForResult()` causes no delay). Using the combinator approach alleviates this issue for developers as the scheduler chains tasks, invoking `waitForResult()` only when the result is immediately available (and thus will not sleep). Realistically, in many cases some code will need to invoke `waitForResult()` on the combinator future, but this is true regardless and the reduction of in-task `waitForResult()`

invocations can improve resource-usage efficiency. See Wojciechowski [124] for a further discussion on combinators as declarative orchestration tools.

2.4.2 Java 8 implementation and background

In this thesis I address Java 8 in particular as it remains the most widely used version of Java. According to JetBrains'[4] Annual Developer Ecosystem Survey, which is based on over 30,000 survey responses, in 2019 Java 8 was used by 83% of respondents that indicated that use some form of Java [46] (when this research began) and 73% in 2021 [47] (the latest results as of writing). Furthermore, large software projects, such as Paterson in our running scenario (Section 1.2.2), are more likely to be on older versions of runtimes (e.g. Java 8). Changing the underlying runtime can be costly from a development perspective (e.g. time-consuming) and comes with a level of risk due to the nature of changing a foundational component of the software.

I describe Java 8's standard task concurrency utilities as a general background, non-standard Java concurrency frameworks may also be applicable. For further documentation and reading, see the Java 8 `java.util.concurrent` package documentation [42].

The core of task-based concurrency programming is the concept of a *future*. A future represents a value that will be available at some point, but not necessarily immediately. For example, the return value of a task. Java 8 includes the `Future` interface with various implementations depending on task implementation. A simple task implementation is the `FutureTask` class. `FutureTask` implements the functionality for executing a task and storing its result, as well as implementing the `Future` interface for accessing, and waiting for, the result. Another implementation of the `Future` interface, that I use in this thesis, is `CompletableFuture` which implements combinator pattern utilities. See Urma et al. [121] for more information on `CompletableFuture`s and combinators in Java 8.

Tasks are created by submitting some executable code, via the `Runnable` or `Callable` interfaces, to an `ExecutorService` via the `ExecutorService.submit()` function which returns a `Future`. `ExecutorService` is the generic concurrency execution interface in Java 8. An example of a task submitted via an `ExecutorService` is given in Fig. 2.2.

The standard implementation of a task in the Java 8 concurrency utilities is `FutureTask`. The general pattern in thread pool implementations of `ExecutorService` (e.g. `ThreadPoolExecutor`) is to create a `FutureTask` with the given `Runnable` or `Callable`, push the `FutureTask` to the task queue, and return the `FutureTask` to the caller of `ExecutorService.submit()` (as the `Future` return value). When a worker thread receives the `FutureTask` to execute, the thread calls `FutureTask.run()` which executes the `Runnable`/`Callable`.

Life-cycle events as Java operations The task life-cycle operations, described above, correspond to `FutureTask` functions as given in Table 2.1.

2.4.3 Related concurrency models

Though this thesis focuses on classical thread-pools, other concurrency models address aspects of (in)efficiency. In this section we will consider two models closely related to

[4]JetBrains is the maker of a range of popular development tools, including the Integrated Development Environment (IDE) *IntelliJ IDEA* [41] and the JVM-based language *Kotlin* [2].

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ExecutorServiceExample {
  static void main(String[] args) throws Exception {
    // Create a thread pool executor service with 8 threads.
    ExecutorService es = Executors.newFixedThreadPool(8);

    Future<?> runnable = es.submit(() -> {
      // No return value.
      System.out.println("Hello task");
    });
    Future<String> callable = es.submit(() -> {
      // Return some value.
      return "Hello world";
    });

    // Wait until the runnable is executed. Will print "Hello task".
    runnable.get();

    // Print the result of the callable task, after waiting for the
    // task to execute. Will print "Hello world".
    System.out.println(callable.get());

    // Output:
    // > Hello task
    // > Hello world
  }
}
```

Figure 2.2: An example Java program that creates two tasks using the `ExecutorService` and `Future` interface, and the `Executors` utility for creating a thread pool `ExecutorService` with 8 threads.

Life-cycle operation	Java function
Task construction	`FutureTask.init()`
Execution start	start of `FutureTask.run()`
Result set	`FutureTask.set()` and variants
Result get start	start of `FutureTask.get()` and variants
Result get completion	end of `FutureTask.get()` and variants

Table 2.1: Mapping of task life-cycle operations to Java `FutureTask` functions.

our target model, Fork-Join [55] and Coroutines [107]. Both models support task-based concurrency and aim, in part, to address the inefficiency of sleeping threads [121].

It is worth noting that unnecessary concurrency (here in the form of creating unnecessary sub-tasks) is worth removing, regardless of the underlying concurrency model, as all forms of concurrency add some amount of overhead. Though, some concurrency models aim to minimise this overhead by using lightweight concurrency primitives so that redundant concurrency has a lesser impact.

When considering related concurrency models, it is also important to take into account that each concurrency model has its pros and cons, none is a perfect solution, and that changing the concurrency model of a piece of software is risky. Changing concurrency models introduces a risk of performance degradation (e.g. via new worst-case behaviour) or even the introduction of new bugs. For example, when a large video game, Fallout 3 [86], was released a significant number of players experienced unexpected crashes. It was discovered that the crashes were due to a concurrency bug. Specifically, the game was developed and tested on machines with 4-core processors and when run on 8-core processors an error could occur and crash the program. Users fixed this bug by limiting the number of cores the program could access [85]. Though an isolated case, this demonstrates the risks and fragility inherent in concurrency and modern software. In short, changing the underlying concurrency model to improve concurrency efficiency would not fit within the bounds of a *small, incremental change* (even if the code change itself were small) – as emphasised in the thesis statement (Section 1.2.1).

Furthermore, it is valuable to have the programmer decide how to fix an issue or improve performance, so though changing the underlying concurrency model may be viable in some instances, it is not necessarily the best course of action. Providing the programmer this choice, and informing them of the effects of making a change, is a key motivation of this thesis.

Section 7.2 discusses how the methods underlying *DTRSO* may generalise to concurrency models beyond than the target concurrency model.

Fork-Join

Fork-Join is a task-based concurrency model based on work-stealing [26, 15, 55, 94]. It aims to improve efficiency by multiplexing tasks across a pool of workers (a worker is a thread within the Fork-Join thread pool) and, most relevantly for this thesis, having workers execute queued tasks while waiting for sub-tasks to finish execution, instead of sleeping. This is one method to address the issue that sleeping threads is harmful (see Section 15.2.3 of [121]).

While this model may initially appear to resolve the same inefficiency of sleeping

that underlies the motivating example of this thesis (Chapter 1), there are a number of reasons why it may not be an appropriate solution (including in situations such as our imaginary scenario, introduced in Section 1.2.2). First, as with all concurrency models, it has drawbacks. The most significant drawback for our motivating example, and our imaginary scenario, is that it has a bad worst-case for latency-sensitive tasks (discussed below). Second, as stated above, modifying the underlying concurrency semantics of an existing program introduces risk, which in many cases will be unacceptable for the benefits achievable.

Fork-Join semantics overview To understand how Fork-Join's worst-case differs from classical thread-pool scheduling, and why it can be bad for latency-sensitive tasks, we must first understand the core of how Fork-Join works[5]. A Fork-Join system manages a pool of workers (in Java, each worker operates on its own thread) and receives task submissions, just as a classical thread-pool does. A task may create sub-tasks (referred to as "forking" a task) to perform concurrent processing. When a sub-task is created, it is added to the task queue of the worker executing the parent task. When the result of a sub-task is required by its parent task it is "joined". The semantics of "joining" is the key point of interest for us. If the sub-task is not being executed, the worker can begin immediately executing it. However, if the sub-task is being executed by another worker and is not yet complete, the worker attempts to find other work to do. The worker will first prefer to execute tasks from its own task queue. If its task queue is empty, it will attempt to *steal* a task from another worker's task queue. Only if there are no tasks in any workers' task queue, will the worker sleep.

Worst-case The semantics of "joining" is where Fork-Join's worst-case differs from that of a classical thread-pool. In a classical thread-pool, when "joining" a sub-task, a thread will sleep until the sub-task is complete, and then immediately continue execution of its current task[6]. This means that a task in a classic thread-pool is only dependent on its explicit sub-task dependencies (as defined by the programmer in the code) – once the sub-task completes, the parent task may continue immediately. However, in a Fork-Join system a task's continuation may become dependent on an arbitrary graph of other tasks, defined at runtime. To make this concept more concrete, imagine a worker, w, executing a task, a, when a attempts to join a sub-task, b, which is being executed on a different worker, w steals another task, x, and begins executing it. Now task a may not continue until task x completes[7]. In the code, a will have an explicit dependency on b defined by the programmer, however, the dependency on x is determined at runtime by the Fork-Join scheduling. Task x can be any arbitrary task and may depend on further sub-tasks. This pattern can repeat indefinitely, creating an arbitrary graph of dependencies for a.

Thus the worst-case for a task in a Fork-Join system is arbitrarily bad. Whereas, the worst-case for a task in a classical thread-pool is sequential execution of all sub-tasks, assuming the thread-pool is configured to schedule tasks on their submitting threads if all threads are currently busy, known as the *Caller Runs Policy* in Java [43]. Of course, a

[5]This is a high-level description of the core of Fork-Join, for a more in-depth description see Lea [55].

[6]Modulo OS-level scheduling of thread execution.

[7]This could be alleviated with continuation stealing, whereby, in this scenario, task a is placed on a "to-be-continued" queue and then executed by a worker (including workers that did not begin the execution of the task) when its dependency (task b) is ready. However, this is not implemented by Java's Fork-Join framework, and comes with its over trade-offs (e.g. context switching costs).

classical thread-pool may also deadlock if thread starvation occurs and the thread-pool policy is to queue all tasks regardless of the state of threads in the pool.

Trade-off Fork-Join thereby trades higher thread utilisation under normal circumstances for an arbitrarily bad worst-case. This trade-off makes sense for certain workloads, but not for others. For example, in a large computational workload where the Fork-Join system is used to compute one result (e.g. a large simulation), the greater thread utilisation is desirable and the worst-case is acceptable as all tasks must complete before the overall result is ready. Whereas, in a latency sensitive system, such as the running scenario (Section 1.2.2) where web requests should be completed as fast as possible so they can be returned to the client, the worst-case is less acceptable as it could lead to long, unpredictable delays based on task scheduling.

Coroutines

Coroutines are a lightweight concurrency primitive generally implemented at the language/runtime level [107]. They can have a similar conceptual design as task-based concurrency, in that they are commonly represented as a function invocation with some set of arguments. Though, they do not necessarily follow the Promise/Future pattern. Their key benefit is their lightweight nature, enabling runtimes to swap their stacks on and off OS threads cheaply. This makes them well suited to I/O bound programs as an I/O operation can block a coroutine, which can then be swapped off a thread, instead of blocking a whole thread.

As an example of a coroutine-based[8] concurrency framework, a key selling point of the Go [28] language is their "Goroutines" which are lightweight coroutines that make it easy for programmers to implement concurrent systems. However, Go does not use the Promise/Future pattern and instead emphasises the use of "synchronisation by communication" via its "channel" data type (which enables two concurrent operations to send messages across a channel) which has inbuilt language support, including special syntax to make it more ergonomic. Much of this design is based off Hoare's seminal work on *Communicating Sequential Processes* [36].

Other implementations do enable task-based concurrency with the Promise/Future pattern. For example, Kotlin [2], a JVM-based language, uses coroutines as the core concurrency construct [24], and C++ 20 introduced support for coroutines [12]. In C++'s implementation, a coroutine is a function which can be suspended and resumed (a synonym for continuation), without blocking a thread. C++ does this by storing, and restoring, the stack information required to resume the coroutine, when the coroutine is suspended. The framework supports the Promise/Future pattern with the `co_await` keyword, which suspends the containing coroutine until the awaited coroutine returns (or yields) a value. This is conceptually similar to Java's `Future.get()`, though without the thread-blocking semantics.

Addressing thread sleeps Coroutine-based concurrency systems are designed to avoid blocking threads by making it relatively cheap (in terms of overhead) to swap coroutines on and off underlying threads. In some ways this is conceptually similar to a Fork-Join

[8] Or "coroutine-inspired", depending on how strictly you define coroutines.

framework with the addition of continuation stealing, meaning the worst-case behaviour described above does not apply.

Due to their benefits, especially their relatively cheap performance cost and ergonomics which make them easy to adopt, coroutines are becoming an increasingly popular form of concurrency (as evidenced by their inclusion in new languages, such as Kotlin, and recent inclusion in existing languages, such as C++). However, other concurrency models, including Fork-Join and classical thread-pools, continue to be widely used and have benefits of their own (e.g. less context switching can improve performance). Furthermore, it would be a significant undertaking to modify a large codebase to use coroutines[9], along with the risks of changing the underlying concurrency semantics described above. So though modern coroutine frameworks are promising options, especially for new software projects, they do not provide a solution within the incremental change approach described in the thesis statement (Section 1.2.1).

2.5 Java and the JVM

The *DTRSO* implementation I develop in this thesis targets Java and the JVM, so at times Java and its semantics are relevant to understanding aspects of *DTRSO*.

As a brief review, Java is a statically typed, object-oriented programming language that is executed in a managed runtime called the Java Virtual Machine (JVM). Java is first compiled into JVM Bytecode and packaged as "class files". These class files are loaded dynamically by the JVM at runtime during "class loading" – when a class's contents is required (e.g. to invoke a method defined by the class) its class file is loaded. The JVM executes JVM Bytecode using a stack machine and provides managed runtime utilities such as garbage collection-based memory management and dynamic exception handling. The JVM also provides a JIT compiler to optimise frequently executed code paths by compiling JVM Bytecode to native code during runtime. As the JVM uses JVM Bytecode, rather than Java directly, there are a number of other languages that are designed to be executed on the JVM (including Scala [79], Groovy [32], and Kotlin [2], among others).

The standard JVM threading model uses logical threads that correlate to OS logical threads. This means they are relatively heavyweight (when compared to coroutine-based systems such as Golang [28]) and as such thread-pool based concurrency is a common pattern.

The JVM is a useful target for *DTRSO* given its type safety, its use of heavyweight logical threads, its support for task-based concurrency in the standard library, and its dynamic class loading enables relatively straightforward instrumentation for tracing.

2.6 Experimental environment

All experiments in this thesis were run on a Linux benchmarking machine with specifications: Intel Core i7-8700 CPU @ 3.20GHz, 32GB RAM, Clear Linux OS [21] 34630. Intel TurboBoost was disabled, the CPU `scaling_governor` parameter was set to `performance`, and hyper-threading was enabled as it provides a more realistic basis for concurrency analysis. Software: Java OpenJDK 1.8.0_292, Rust 1.50.0-nightly (*Quilt*, the tracer

[9]Not to mention the codebase must be written in a language that supports coroutines.

introduced in Chapter 3, is implemented in Rust). Experiments were run using Docker [68] containers[10], Docker version 20.10.6.

2.6.1 *Acme* – Real-world evaluation target

Throughout this thesis I use a real-world industrial API server, which I refer to as *Acme*, to evaluate methods and perform experiments. *Acme* provides an analogue of the Paterson software from our running scenario (Section 1.2.2). Use of a real-world system for evaluations and experiments is essential to demonstrating that the problems and questions I investigate exist in the real-world and the methods I describe are effective at addressing them. *Acme* has a number of specific benefits as an evaluation system:

- It conforms to the target concurrency model, as described in Section 2.4.
- It has been in production use for approximately 5 years at the time of writing.
- It has extensive automatic test coverage, allowing us to confirm that no changes to its source-code affect functional program behaviour.
- It is actively maintained and developed, as per the thesis statement (Section 1.2.1), so it is not a legacy system that has been discarded due to design faults.

Technical properties

To give a better sense of the standard operation of *Acme*, I detail its technical properties such that the reader may have a general sense of the software.

At a high-level, *Acme* is the server "backend" for a user-facing application. Its operations range from cryptographic authentication and database reads and updates, to image processing and data streaming.

Its workload at any given time is defined by the API requests made to its "endpoints" (including, for example, user registration, image upload, loading of data analytics, etc). Though each endpoint performs a unique function (in the product sense), they broadly perform similar underlying technical operations. Indeed, most of the functionality of *Acme* is implemented in classes and libraries used by multiple endpoints (i.e. many endpoints will invoke similar call-trees). For example, all endpoints that require authentication will invoke a shared `checkAuthenticationHeaders()` function which will perform some cryptographic and database operations to ensure the request is appropriately authenticated. Moreover, all endpoints follow the same high-level pattern of: receive request, create tasks on the thread-pool, compute the request response (using the results of the tasks), and return the response to the client.

Thus the key behaviour of *Acme*, for the purpose of this thesis, is defined by the tasks performed on the thread pool. The general groupings of these tasks are:

- I/O-bound tasks include:
  - Database queries, which require a network request.

[10]Docker containers simplify benchmark environment management and consistency. On Linux, Docker containers have been shown to have negligible impact on CPU and memory speeds [25] (unlike virtual machines).

  - Remote service requests (e.g. sending notifications, sending automated emails, integrating with third-party API services, enqueuing items on a distributed queue, etc), which also require a network request.

- Compute-bound tasks include:
  - Server-side javascript execution.
  - Computation of domain-specific analytics.
  - Cryptographic operations (e.g. secure password hashing with Bcrypt [92] and encryption of data blobs).
  - Graph traversal.
  - Image processing (e.g. resizing, compressing, reformatting, etc).

- Tasks that are a mix of computation and I/O include:
  - System synchronisation tasks (e.g. translating events from a distributed cluster into client-consumable data and then streaming that data to connected clients).
  - Coordination tasks which primarily start sub-tasks and combine their results. These tasks may also perform computation of their own.

Some endpoints are I/O bound (e.g. those that primarily perform a series of database queries) whilst others are computationally bound (e.g. those that perform image processing). Most endpoints, though, are a mix of I/O and computational work. Similarly, most endpoints generate between 10 and 100 tasks during their execution, though some outlier endpoints may generate more (e.g. analytics tasks that perform a large number of database requests).

Evaluation properties

While using a proprietary system for experiments and evaluations reduces transparency of the results, I believe it is worthwhile for the improved efficacy of the experiments and evaluations. In particular, in this thesis we are interested in evolving large existing systems to improve performance (as per the thesis statement, Section 1.2.1), in this context *Acme* can be thought of as an analogue of Paterson from our running scenario (Section 1.2.2). Moreover, as stressed in the introduction, this thesis focuses on the "theoretical development of a practical technique". By "theoretical development" I mean that the key contributions are the methods and their theoretical underpinnings, not the quantitative evaluations. Whilst "practical development" means that those methods must be demonstrated to be truly practical, efficacious, through evaluations (quantitative and qualitative).

A key metric we focus on in evaluations is *throughput*, improvements in throughput demonstrate a clear improvement due to an optimisation. It is worth understanding the difference between throughput and *latency*. Latency is the time between asking for something and receiving the answer. Throughput, on the other hand, is the amount of work that can be processed by the system while under load. Thus, an increase in throughput means the system can process more items of work in a given time while under load, though the latency of each item may not necessarily change. In the case of *Acme*, throughput relates the number of API requests that can be processed concurrently (which is bounded by system resources and, in particular, the size of the thread pool).

Chapter 3

Execution Tracer

The first component of *DTRSO* is the execution tracer, *Quilt*, described in this chapter. *Quilt* instruments a program and records its execution as a *trace-log*, which is then used by *Rehype* (Chapter 4) to identify improvements. Chapter 6 further augments the tracer to investigate the effects of tracing-overhead on program concurrency behaviour.

Narrative instalment

In our imaginary scenario (Section 1.2.2), Banjo has decided that, to improve Paterson's performance, they will reduce the number of threads spending unnecessary time sleeping (thereby reducing the thread pool saturation rate). Banjo must first determine what code these threads are executing that is causing them to spend so much time sleeping. To understand what Paterson's concurrency and performance behaviour is, Banjo must use dynamic analysis, as both forms of behaviour are exhibited at runtime. This requires capturing runtime data while executing Paterson. To do this in a controlled manner, Banjo executes an artificial workload on Paterson, while tracing Paterson using *Quilt*.

Quilt will generate data (a trace-log) that will then be analysed in the next chapter to identify and quantify potential improvements to Paterson's performance.

3.1 Introduction

Capturing high quality data is essential to enabling high quality program analysis. Capturing it relies on identifying the right data for the analysis to be performed and, in the case of performance analysis it, capturing it without significantly impacting the performance and execution of the target program. As we will see in Chapter 6, tracing introduces overhead and therefore, naturally, has an observer effect (the act of tracing perturbs the program being traced and thus the trace data captured). Thus, to perform accurate program analysis in this domain requires a low-overhead tracer (*Quilt*'s target overhead is discussed in Section 3.1.1).

In this chapter, I present a low-overhead tracer, *Quilt* (Quiet User-space-Instrumenting Lockless-Tracer), that instruments Java programs at runtime, captures program behaviour events, and writes those events to disk as a *trace-log*. The tracer takes as input a *trace-config* which describes which aspects of the program to trace. During instrumentation the tracer intercepts the JVM class-loading pipeline and modifies class bytecode inline to insert tracing code. Performing instrumentation at runtime allows the tracer to instrument

all (JVM bytecode) functions, not just those available as source code. The tracing code calls an in-process utility which records the traced events to disk. To minimise the impact of tracing overhead, the tracer (Section 3.4) uses a novel lockless algorithm to record events without blocking the target program's threads. This algorithm trades memory for locklessness – it uses more memory (in the form of per-thread buffers) to enable lockless tracing and thereby avoid blocking threads.

The trace-log (Section 3.2) contains a sequence of events (such as function entry and exit, task lifecycle, and thread pool administration events) to represent program behaviour. The events captured are designed to enable tracing concurrent execution across thread boundaries and, equally importantly, also provide enough program detail to determine, in subsequent analysis, which part of the source code must be modified to implement a suggested change. Knowing all of the times a program's execution crossed thread boundaries is not especially helpful if you do not know which source code caused it.

The events captured trade specificity for lower instrumentation and tracing overhead. For example, it traces function entry and exit events instead of call-site events. This means that the call-sites must be derived during analysis, which is not always possible. However, the performance impact is substantial as, firstly, far fewer locations need to be instrumented, and secondly, there are fewer unique locations to be tracked so each event can be stored with fewer bits. Since the tracer instruments the program during runtime, the performance of instrumentation is relevant to the overhead of the tracer. Similarly, the tracer does not capture all possible events (i.e. it generates sparse traces), instead it captures events as defined by the *trace-config*, which are intended to enable subsequent analysis to generate useful results while minimising the tracing performed. Capturing sparse traces introduces difficulties in subsequent analysis phases as they have less precise data to work with. We discuss these issues further in Section 3.3.

While some other tracing approaches specifically capture "non-deterministic" events (e.g. filesystem IO) to enable record & replay systems (e.g. Castor by Mashtizadeh et al. [66]), such an approach is not suitable for the performance analysis we consider in this thesis. With regards to performance, all events are naturally non-deterministic. Indeed, even simple functions may take varying amounts of time to execute given the same input, as the broader system environment may be different. This is especially true for concurrent programs as performance of one thread is directly affected by the resource usage of other threads[1]. So tracing even otherwise deterministic events becomes important to accurately analysing the performance effects of potential changes.

Managed runtimes, and the JVM in particular, make it significantly easier to instrument and trace programs in a fast and safe way. Though native programs can be similarly instrumented, there is less available tooling and it is easier to introduce errors and inefficiencies. In particular, managed runtimes enable runtime instrumentation (e.g. during class loading), provide debug symbols (e.g. function signatures) by default, and (almost) automatically integrate inserted tracing code (e.g. JVM bytecode) into existing error handling and JIT compilation. Moreover, instrumentation tools can take advantage of greater runtime information (e.g. program state) to decide when and how to instrument and trace code.

Experimental results (Section 3.5) demonstrate the tracer introduces low overhead (in

[1]Of course, a sceptic could argue that other processes could also be affecting such variations, which is true. However, we must accept such potential disturbances to traced data as a natural part of real-world environments.

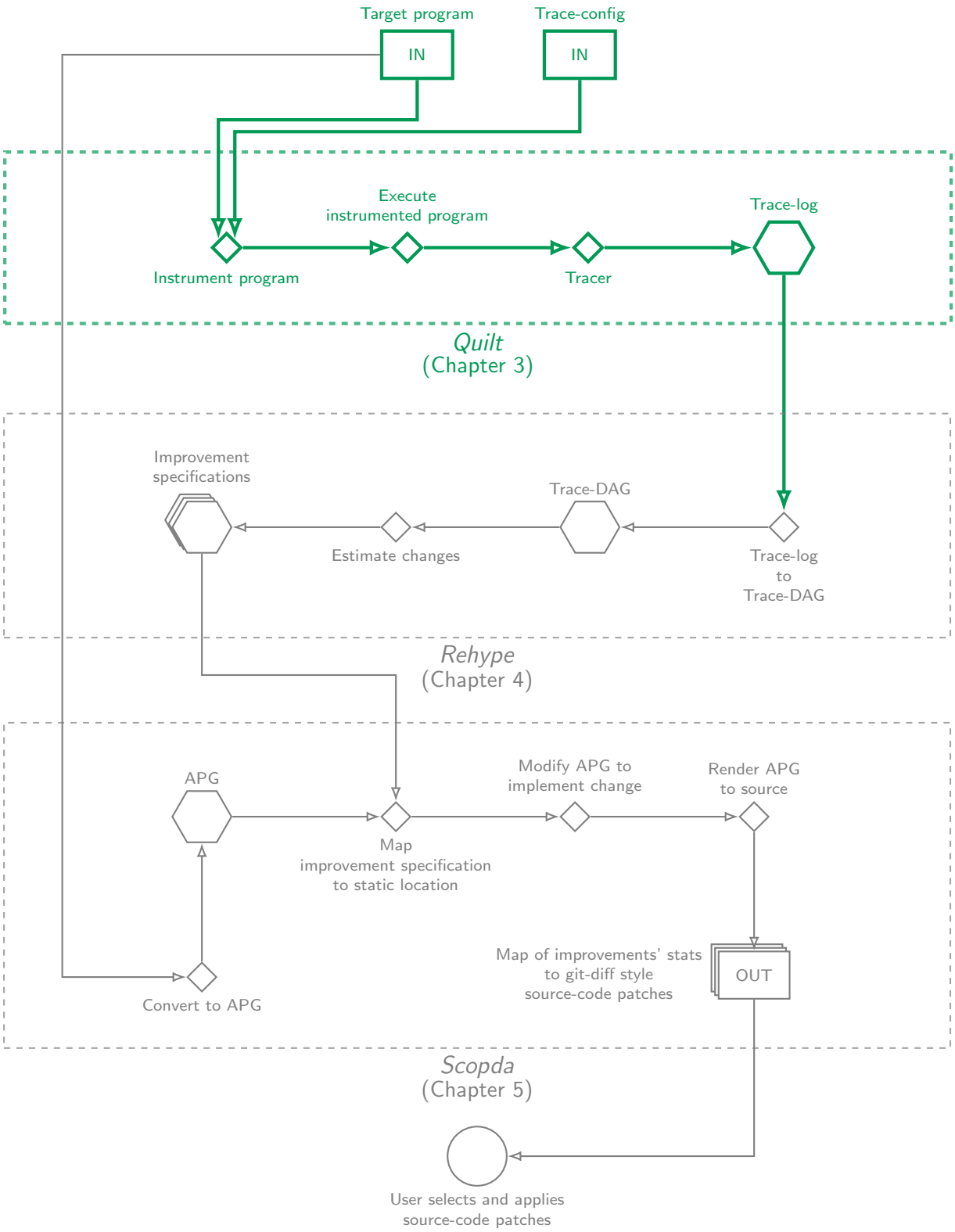


Figure 3.1: High-level overview of where *Quilt* exists within *DTRSO*. *Quilt* takes in a *trace-config* and the target program, instruments the program, executes the program while tracing it, and generates a *trace-log* which is used by *Rehype* (Chapter 4).

Tool	Overhead range (magnitudes)	Source
Sampling profilers	Overhead for 1KHz sampling	
Hofer et al.	1 – 10%	[37]
`perf`	1 – 10%	[76]
SPS & ISPS	1 – 10%	[38]
`tgp`	1 – 10%	[96]
`jvmti`	10 – 100%	[38]
Tracers		
Nisbet et al.'s `bcc`-based thread tracer	1 – 10%	[76]
Hofer et al.'s lock tracer	1 – 10%	[39]
Castor (record & replay)	1 – 10%	[66]
Intel-PIN	10 – 10,000%	[6]
Execution control		
Valgrind	100 – 10,000%	[74]

Table 3.1: Overhead order-of-magnitude ranges for various runtime data capture tools, including samplers, tracers, and execution control tools. This table does not intend to comprehensibly report on all existing tools, but rather give a general perspective on the orders of magnitude of overhead incurred by a variety of tools.

the order magnitude range of 1 – 10%). Finally, I position this chapter's work among related work in Section 3.6 and conclude in Section 3.7.

3.1.1 Target overhead

A key aim of *Quilt* is to incur as little overhead as possible during tracing. As we will see in Chapter 6, overhead has a direct impact on the quality of subsequent analysis, as overhead can distort the behaviour of a program. *Quilt* aims to incur overhead in the order of magnitude range of 1 – 10% of the wall-clock time of a program (though this is a limited indicator of overhead, as discussed in Chapter 6). To put this range in context, Table 3.1 lists a number of existing tools and the order of magnitudes of their overhead (according to existing literature).

As with all tracers, the amount of overhead *Quilt* incurs is dependent on the amount of data (i.e. events, in the case of *Quilt*) captured and the rate at which it is captured. Therefore, the two primary levers for controlling overhead are the overhead per-event and the number of events captured. While *Quilt* can be configured to capture fewer events and thus reduce overhead (see Section 3.5), there is a necessary minimum set of events required to enable subsequent analysis by *Rehype* (Chapter 4). Hence, minimising per-event overhead is a key aim of the methods and optimisations in *Quilt*.

3.2 Trace-log

The trace-log is a series of events captured during tracing. It is the primary output of the tracer and is the primary input to the analyser (Chapter 4).

There are three categories of events: function, task life-cycle, and thread pool events. All events are listed in Table 3.2.

All events contain a type byte, a nanosecond timestamp of their occurrence, and the thread-id (see Section 2.4) of the thread they occurred on[2]. The function and task events have extra associated data:

Function: events also contain the id of the method they occurred in. Method ids correspond to a *method-index* file generated during instrumentation, however, this is largely an implementation detail.

Task life-cycle: events also contain the task-id of the task they relate to (recall from Section 2.4, task-ids are assigned at task construction and are unique within a trace-log).

Thread-pool: events have no auxiliary data.

A C-style definition of the event data is given in Fig. 3.2

While the start of a task's execution is explicitly traced (`TkExeStart`), the end of a task's execution is recorded as the setting of the result of the task (`TkResSet`) in the task data structure (e.g. a `FutureTask`). This is because the concurrent users of the task (i.e. threads emitting `TkResGetStart`/`TkResGetEnd` events) wait for the result of the task, not for the full completion of execution. Thus recording the event that enables other threads to continue their work is relevant to concurrency analysis, while recording the completion of execution is not (though they may, in some cases (e.g. `FutureTask`), be effectively the same).

When a thread finishes executing a task it will always emit a `ThreadWait` event. This occurs irrespective of whether a another task is available to execute as the thread pool only assigns tasks to threads that are in the *unoccupied* state.

We assume a form of sequential consistency: that events executed by a single thread appear in event order, and with strictly increasing timestamps, in the *trace-log*; events from separate threads may interleave in the *trace-log*, not necessarily in timestamp order, but events can be re-ordered in post-processing to not interleave or to be interleaving and sorted in timestamp order. Furthermore, events of different types may interleave in the *trace-log*.

3.3 Sparse tracing

An important consideration of the tracer is its sparse tracing approach. While sparse tracing is a natural aspect of real-world tracers, it significantly affects subsequent analysis and introduces non-trivial complexity.

To understand the effects of sparse tracing, it is useful to first understand the alternative, full tracing. A simple definition of full tracing, for our purposes, is tracing that generates a

[2]These fields are the logical fields of each event. In practice *trace-log* size is optimised by storing pages of events from threads with one thread id per page.

Event	Location
Function events	Events inserted into methods defined by the *trace-config*. Auxiliary data is a 'method id' – a unique identifier of the method the event occurred in.
`FnStart`	First bytecode instruction in a method.
`FnEnd`	Bytecode instruction immediately before return instructions in a method.
`FnCatch`	Bytecode instruction at the start of `catch` blocks in a method.
Task life-cycle events	Events specially inserted into task class(es) (`java.util.concurrent.FutureTask`). Auxiliary data is a 'task id' – a unique identifier for each instance of a task, this is assigned to a task during construction.
`TkCtor`	In the constructor of the task class. This event is also annotated, during processing, with the stack of active traced functions on the thread it occurs on. This is used when outputting suggested source changes. See Section 4.2.3.
`TkExeStart`	At the beginning of the task class execution method.
`TkResSet`	At the start of the task class result set. This event also signals the end of the task's execution.
`TkResGetStart`	At the beginning of the task class get result method(s).
`TkResGetEnd`	At the end of the task class get result method(s).
Thread pool events	Events specially inserted into thread pool worker thread class(es) (`java.util.concurrent.ThreadPoolExecutor`). No auxiliary data.
`ThreadWait`	When a thread in a thread pool enters the *unoccupied* state. This will occur when a thread is initially constructed and when the thread finishes executing a task.
`ThreadStart`	When a thread is assigned a task to execute (i.e. the thread transitions state from *unoccupied* to *active*).

Table 3.2: Listing of all event types traced by *Rehype*.

```
struct Event {
  // The type of event (e.g. FnStart, FnEnd, etc).
  enum EventType type;

  // The nanosecond time when the event occurred.
  uint64 timestamp;

  // The id of the thread that this event occurred on.
  uint32 thread_id;

  union {
    // Function events contain an id of the method they occurred in.
    uint32 function_event_method_id;

    // Task life-cycle events contain the per-instance id of the task
    // they relate to.
    uint32 task_id;

    // Thread pool events do not contain extra data.
  } aux_data;
}
```

Figure 3.2: A C-style definition of the data for each event.

```
void caller(boolean x) {
  if (x) {
    callee1();
    callee2(3);
  } else {
    callee2(5);
  }
}
```

Figure 3.3: Example of function that calls another function (`callee2`) in two separate caller-contexts. As the present tracer does not capture call-site information, this can lead to an ambiguity in trace-logs that subsequent analysis must deal with.

"trace that captures the exact operations performed, and their timing, such that subsequent analysis has full information about how the program executed". Note the difference between "how" the program executed and "what" the program executed. The latter can be achieved using record/replay systems and recording just non-deterministic events, the former requires tracing the behaviour of the program during execution.

Technically, pure full tracing would involve tracing every CPU instruction. Pure full tracing, though, provides an unnecessary level of detail; moving up the OS stack, tracing every branch and language-level instruction could be considered full tracing. While having full tracing would make some aspects of analysis fairly straightforward, as there is no uncertainty, it would introduce far too much overhead, to the point that subsequent analysis would be invalid due to tracing distorting the program behaviour (Chapter 6).

As such, it is practically necessary to limit what aspects of the program are tracked, resulting in sparse tracing. Sparse traces are simply traces that do not capture full information about the execution of the program. For example, the present tracer does not capture branch statements or parameter values. Similarly, it does not track call-sites, but rather function entry and exit events.

Not capturing call-site information limits the function events' *caller-context*. Specifically, they cannot disambiguate multiple call-sites with in a function invoking another function. For example, in Fig. 3.3 the `caller` function calls `callee2` in two separate caller-contexts. If some analysis needed to differentiate between these caller-contexts (e.g. to make a change to one call but not the other, such as changing the parameter value), it would have to derive such information from the broader context (e.g. whether the trace contains a call to the `callee1` function immediately before a call to `callee2`). This is effectively what the analyser in Chapter 4 and the source code patch generator in Chapter 5 have to do.

It is clear, then, that sparse traces is a trade-off between tracing overhead and analysis complexity. Due to the significant impact of tracing overhead, this is a trade-off we accept. Though it is worth being aware of the effects of such a trade off.

3.4 Tracer

Quilt is an execution tracer that takes in a *trace-config* and generates a *trace-log*. The trace-config specifies which functions of the target program should be traced. *Quilt* generates function and concurrency framework events (e.g. task-spawn events). The vast majority of events generated are function-entry and -exit events. These contain a nanosecond

time-stamp, a function identifier, and id for the thread that the event occurred on.

Quilt consists of two primary components, the *instrumenter* and the *tracer*. The instrumenter inserts tracing code based on a trace-config. The tracer is an in-process utility that receives calls from tracing code and records corresponding events on disk as a trace-log.

The instrumenter intercepts class loading within the JVM (Fig. 3.4a) and modifies class bytecode to insert tracing code (Fig. 3.4b). For each class file intercepted, the instrumenter checks whether any method should be instrumented according to the trace-config. Each such method is then registered with a *method-index*, which maps method signatures to unique method IDs (which are stored in the trace-log), and then the method's bytecode is rewritten to include the tracing code (calls to the in-process tracer's functions). Finally, the class file is reconstructed with the updated bytecode and returned to the JVM class loading process. The instrumenter's architecture and instrumentation process is illustrated in Fig. 3.4. An example of a generic method's bytecode pre- and post-instrumentation is given in Fig. 3.5. For specialised instrumentation, such as task-lifecycle tracing code, event-specific logic is used to determine where to insert the appropriate instructions within the target method (e.g. variations of `ExecutorService.submit`). Such specialised logic exists within the same overall process described above.

The tracer is designed for minimal overhead. It prioritises minimising time spent on the target program's worker threads. To achieve this, it uses a client-server architecture with a single server and one client for each JVM thread. The clients generate logs of per-thread events and the server writes these to disk on a background thread. Logs are transferred using a novel *lockless buffer exchange protocol*, to ensure the target program's worker threads do not block due to tracing.

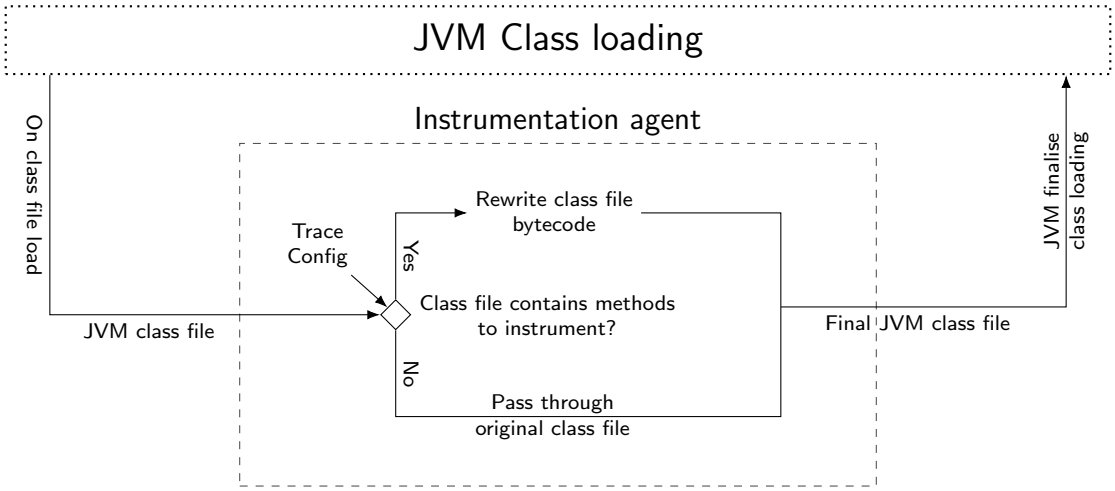
3.4.1 Lockless buffer exchange protocol

The key purpose of this protocol is to avoid blocking operations on the traced program's working threads. This means that all IO operations must be performed on a background thread (hence the server performing all IO operations). The protocol also aims to minimise memory usage.

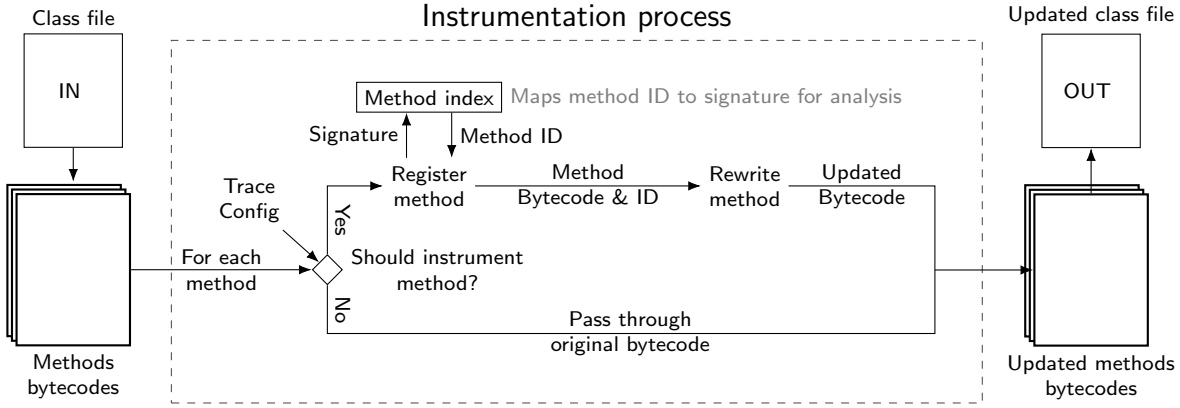
Here we provide a high-level description and intuition for the protocol, Section 3.4.1.1 provides a more concrete specification of the protocol. Fig. 3.6 visually illustrates the client and server as state-machines and Fig. 3.7 provides an example sequence diagram.

Each client uses two buffers, the *active* and *inactive* buffers. Each buffer has a boolean flag that indicates whether the client or the server has *control* of the buffer. The *active* buffer is always controlled by its containing client. Clients write new events to their *active* buffer. When a client's *active* buffer fills, the client attempts to swap its *active* and *inactive* buffers. If a client controls its *inactive* buffer, the buffers are swapped and the client sets the new *inactive* buffer to be controlled by the server. However, if a client's *inactive* buffer is controlled by the server when its *active* buffer fills, the client will expand its *active* buffer (using standard buffer expansion, i.e. doubling the buffer capacity). This expansion step does not require locking as the server will never access the *active* buffer directly.

The server runs a periodic loop checking for client *inactive* buffers flagged as controlled by the server, when it finds one, it writes the buffer to disk. The server uses two further techniques to minimise occurrences of clients expanding their *active* buffers. First, the server can request specific clients swap their buffers before their *active* buffer reaches



(a) A high-level overview of how the instrumentation fits within JVM class loading. The instrumentation agent intercepts JVM class file during class loading, checks if it contains any methods that should be instrumented according to the *trace-config*, if it does the agent rewrites the class file and passes it on to the next stage in class loading, if the class file does not contain any such methods, it is passed to the next stage directly.



(b) An overview of the instrumentation process applied to class files that contain methods to be instrumented. Each method is first tested against the *trace-config* to determine if it should be instrumented. If it should be instrumented, it is first registered with the *method index*, which returns a unique *method ID*. Then, using this ID, the method bytecode is rewritten to include the tracing code. The rewritten bytecode is then combined with the other methods' bytecode to generate the final output class file.

Figure 3.4: A high-level overview of how *Quilt* instruments programs during execution by intercepting JVM class loading.

```
 0 aload_0
 1 invokevirtual #1
 4 ifnonnull 7
 5 aload_1
 6 areturn
 7 aload_0
 8 areturn
```

```
+  0 ldc_w <n>
+  3 invokestatic #<tracer start function>
   6 aload_0
   7 invokevirtual #1
  10 ifnonnull 19
  11 aload_1
+ 12 ldc_w <n>
+ 15 invokestatic #<tracer stop function>
  18 areturn
  19 aload_0
+ 20 ldc_w <n>
+ 23 invokestatic #<tracer stop function>
  26 areturn
```

Figure 3.5: An example of a simple method's bytecode before and after instrumentation. The instrumenter inserts a call to the tracer start function (an `invokestatic` instruction) at the beginning of the bytecode and one call to the tracer stop function before each return instruction. Each call instruction is preceded by an `ldc_w` instruction which pushes the unique method id to the top of the stack.

capacity. When a client generates an event, it checks its swap-request flag, if set, the client immediately swaps its buffers. Second, the server has an additional *interchange* buffer which is exchanged[3] with a client in place of its *inactive* buffer when the server takes the *inactive* buffer. This enables clients to swap their buffers, even if the server is still writing their *inactive* buffer to disk.

3.4.1.1 Algorithmic specification

Given the intuition given above, we now turn to a concrete algorithmic specification of the protocol. This subsection aims to provide a reference for deeper understanding and implementations. However, it does not introduce new concepts or intuition and so may be skipped without hindering understanding of subsequent content.

The protocol consists of three data structures, *client*, *server*, and *lockless-buffer* (LB), and two algorithms, client-side (Alg. 1) and server-side (Alg. 2). The data structures contain:

Client: A triple: {`boolean: requestSwapFlag, LB: activeBuffer, LB: inactiveBuffer`}.

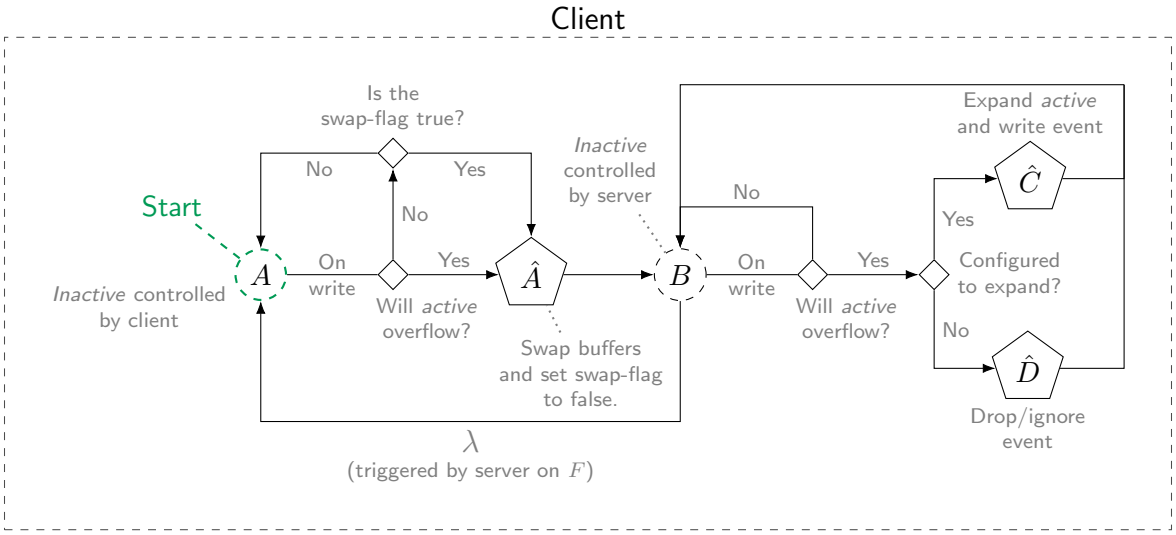
Server: A pair: {`list of clients: clients, LB: interchangeBuffer`}.

Lockless-buffer: A pair: {`heap pointer: dataBuffer, control state: controlFlag`}. The `controlFlag` indicates whether the LB is ready to be used by the client or the server with the values `CLIENT_CONTROL` and `SERVER_CONTROL`, respectively.

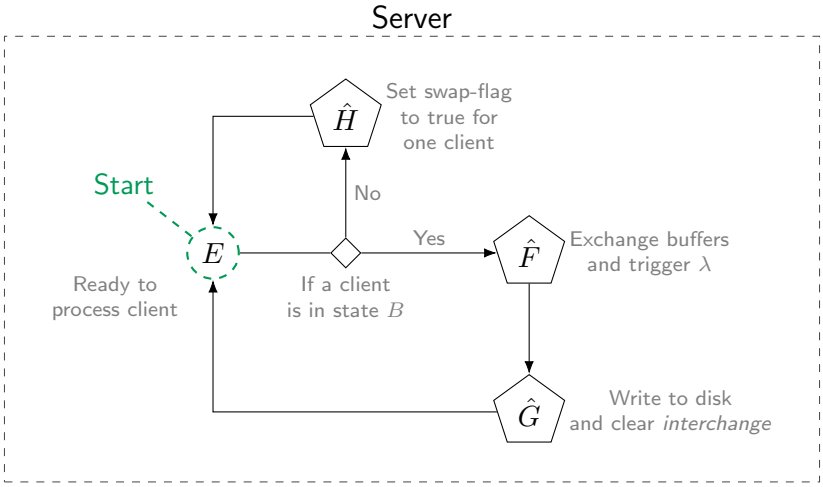
Clients and servers pass control of LBs via the LB's control-flag but never attempt to take control, enabling the lockless exchange. Clients always have control of at least one of their associated LBs.

When a client receives an event it will try to store it in its `activeBuffer`. If it is full, the client checks if the `inactiveBuffer` is in `CLIENT_CONTROL` state. If so, the client sets its

[3] *Swapping* refers to the client swapping its *active* and *inactive* buffers, while *exchanging* refers to the server exchanging its *interchange* buffer with a client's *inactive* buffer.



(a) A state-machine representation of the client within the lockless buffer exchange protocol. The client begins in state A with two empty buffers, *active* and *inactive*. Each time a trace event is received, the client checks whether writing the event will cause the *active* buffer to overflow. If it will, or if the client's *swap-flag* is true, the client swaps its buffers and sets the *swap-flag* to false ($\hat{A}$) and moves to state B. When a trace event occurs while the client is in state B, the same overflow check is performed, except that when true, the *active* buffer must be expanded, or the event must be dropped, and the client remains in state B. Finally, the client returns to state A when the server triggers the λ edge by exchanging buffers with the client.



(b) A state-machine representation of the server within the lockless buffer exchange protocol. The server runs a continuous loop checking for a client whose *inactive* buffer is set to server control. When one is found, it exchanges the buffers, triggers the λ state edge to move the client back to A, writes the filled client buffer to disk, clears the buffer (as it becomes the server's new *interchange* buffer), and resets. If no client is ready, the server sets the *swap-flag* on one client to request it prematurely swaps its buffers, allowing the server to perform under more consistent load.

Figure 3.6: Illustrations of the lockless buffer exchange protocol client and server as state machines.

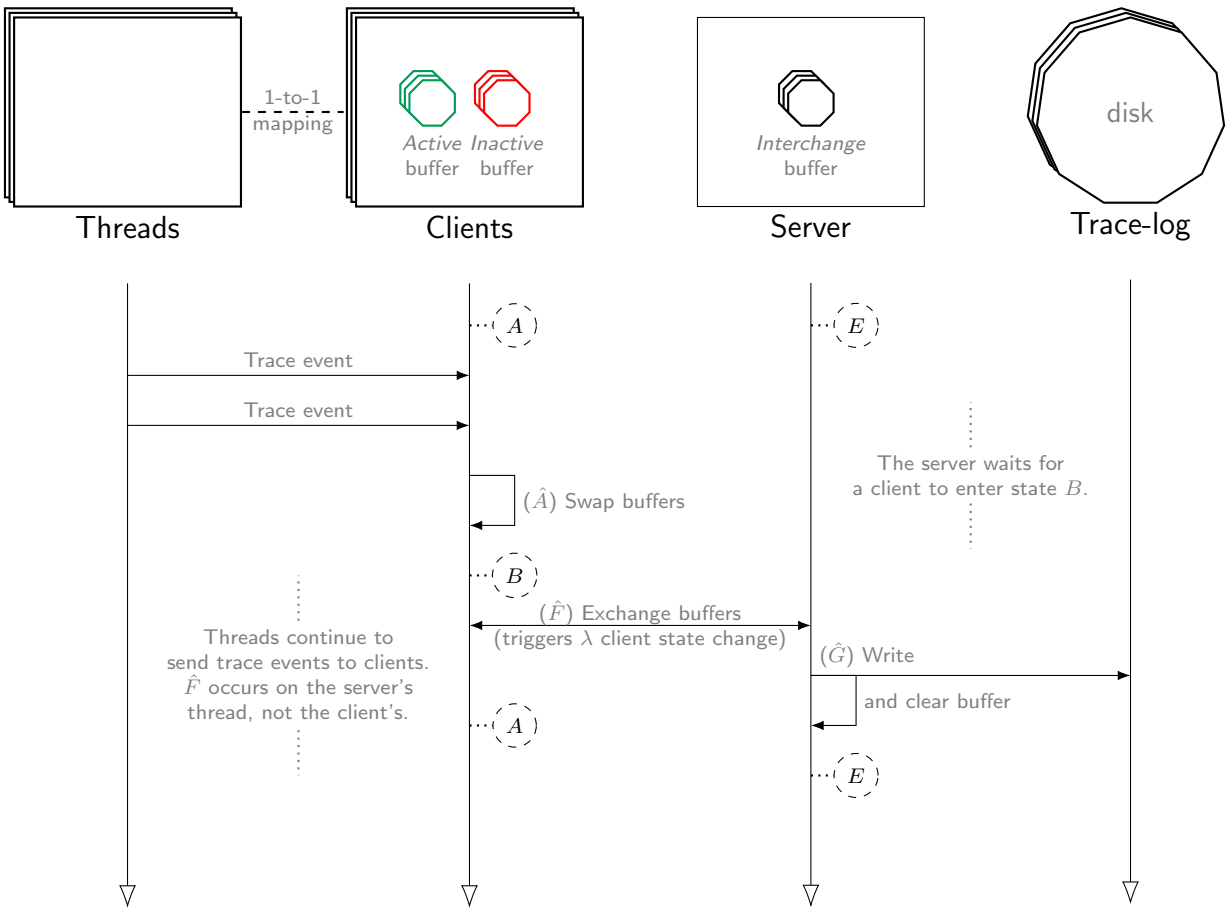


Figure 3.7: An example timeline of the lockless buffer exchange protocol. A thread traces events to a client, which then swaps its buffers. The server then exchanges buffers with the client (triggering the λ state change), then writes the buffer to disk, and clears the buffer. The dashed circles indicate the current state of the client/server (see the state-machines in Fig. 3.6).

Algorithm 1: The client-side algorithm for the *LBSP*. Executes when an event occurs in the traced code.

```
1  Function swapBuffers(c: Client) is
   |  /* Set the first buffer to SERVER_CONTROL so the server will
   |     clear it.                                                  */
2  |  c.activeBuffer.flag ⟵ SERVER_CONTROL;
3  |  temp ⟵ c.inactiveBuffer;
4  |  c.inactiveBuffer ⟵ c.activeBuffer;
5  |  c.activeBuffer ⟵ c.inactiveBuffer;
6  end

   Data: c;                                  /* Thread-specific client.  */
   Input: e;                                        /* Event to trace.  */

   /* Check if the server has requested this client swap its buffers.
      */
7  if c.requestSwapFlag is true AND c.activeBuffer is not empty then
8  |  swapBuffers(c);
9  end

   /* If the first buffer has reached capacity try to swap it with the
      second buffer.  If the second buffer is SERVER_CONTROL expand the
      first buffer.                                                  */
10 if c.activeBuffer.length + sizeof(e) > c.activeBuffer.capacity then
11 |  if c.inactiveBuffer.flag is CLIENT_CONTROL then
12 |  |  swapBuffers(c);
13 |  end
14 |  else
15 |  |  expand c.activeBuffer to a minimum size of c.activeBuffer.length +
   |  |   sizeof(e);
16 |  end
17 end
18 write e to end of c.activeBuffer;
```

`activeBuffer` to `SERVER_CONTROL` and makes is `inactiveBuffer` the `activeBuffer`. Otherwise, if the `inactiveBuffer` is in `SERVER_CONTROL`, the client will expand its `activeBuffer`. Finally, the client writes the event to its `activeBuffer`. This is described in Alg. 1 lines 10 to 18.

The server operates a continuous worker loop in a dedicated background thread. Each iteration the server checks for a client with an LB in `SERVER_CONTROL` state. If found, the server takes the LB and gives the client the server's `interchangeBuffer` (i.e. the "buffer swap"). The server then writes the contents of the LB to disk, sets the LB's flag to `CLIENT_CONTROL`, and moves to the next iteration of the worker loop. This is described in Alg. 2 lines 2 to 12.

If multiple clients set buffers to `SERVER_CONTROL` at the same time, a backlog can develop. This can lead to clients expanding their `activeBuffer`s as they reach capacity before the server has processed the backlog.

To avoid backlogs, the server actively seeks work. If on an iteration the server does not find an LB in `SERVER_CONTROL`, it locklessly requests a client to pre-emptively change buffers (i.e. before its `activeBuffer` fills) (Alg. 2 lines 13 – 18). This work seeking approach smooths the server workload, avoiding backlogs, reducing buffer expansions, reducing tracing overhead.

The initial LB buffer size is set by a parameter in the *trace-config*. It can be beneficial to tune it to minimise both buffer size and buffer expansions.

3.5 Evaluation

We evaluate tracer overhead against *Acme*, an industrial Java server-backend API application (Section 2.6.1; *Acme* is also used in evaluations of *DTRSO* in later chapters), and the Dacapo benchmark suite by Blackburn et al. [8] (specifically, the Lusearch benchmark). As discussed in Section 3.1.1, *Quilt*'s overhead is determined by the number of events captured and the per-event overhead incurred in tracing those events. The *Acme* evaluation provides a realistic assessment of the number of events captured and overhead introduced for normal analysis (we use the same trace-config used for generating suggested source changes in the following results sections). The Dacapo benchmark provides an evaluation target for testing the overhead at the limits (i.e. testing the per-event overhead by tracing a large number of events). For the Dacapo benchmark we use two trace-configs, one that traces all Java functions (including standard library functions; generating over 600,000,000 events), and one that traces only functions relevant to the benchmark (i.e. Lucene package functions). The latter enables us to evaluate *Quilt*'s normal overhead on an open benchmark. As with other chapters, these experiments were run on a Linux benchmarking machine described in Section 2.6.

We evaluate tracer overhead as *user-time* clock and *wall* clock overhead. The user-time clock duration of an execution is the amount of processor time used by the execution. This captures the effect of tracing in multiple threads, whereas wall clock duration can lessen the apparent overhead if concurrency is effectively used. Moreover, the effect of the tracer's memory usage can be captured, imperfectly, via user-time clock overhead as any cache or paging issues caused by the tracer's memory usage will be reflected in the user-time clock overhead. This holistic approach to calculating overhead also ensures the effect of the buffer-server is incorporated into overhead (as it takes processor time, parallel to the target program's execution).

Algorithm 2: The server-side algorithm for the *LBSP*.

```
   Input: ClientList ;                                  /* A list of clients.  */
   Input: interchangeBuffer ;                           /* A Lockless Buffer.  */
 1 while true do primary_loop
 2     foreach c in ClientList do
 3         if c.inactiveBuffer.flag is SERVER_CONTROL then
               /* Exchange buffers with c.                                       */
 4             temp ⟵ c.inactiveBuffer;
 5             c.inactiveBuffer ⟵ interchangeBuffer;
               /* At this point c can use its inactiveBuffer when its
                  activeBuffer reaches capacity.                                 */
 6             interchangeBuffer ⟵ temp;
 7             write interchangeBuffer to trace-log on disk;
 8             interchangeBuffer.flag ⟵ CLIENT_CONTROL;
               /* Move processed clients to the back of the list.               */
 9             move c to back of ClientList;
10             continue primary_loop;
11         end
12     end
       /* No clients' inactiveBuffer was SERVER_CONTROL so the server
          requests a client swap its buffers when it next receives an
          event (i.e. the server seeks work).                                    */
13     foreach c in ClientList do
14         if c.requestSwapFlag is false then
15             c.requestSwapFlag ⟵ true;
               /* Sensible implementations will implement a sleeping
                  back-off to conserve resources.  Clients will not swap
                  their buffers immediately.                                     */
16             break;
17         end
18     end
19 end
```

Program	# Events traced	Mean per-event overhead (ns)
Acme	$\sim 750,000$	~ 1500
Dacapo – Lusearch	$\sim 630,000,000$	~ 100

Table 3.3: The overhead incurred by the tracer on executions of *Acme* and Dacapo's Lusearch benchmark Blackburn et al. [8]. Overhead times are user-time clock durations. These results indicate the low per-event overhead incurred by *Rehype*, with a mean per-event overhead of ~ 100 nanoseconds on Dacapo. The mean per-event overhead is calculated as the total overhead divided by the number of events traced. As such, the cost incorporates the base overhead introduced by *Rehype* (e.g. during setup on the tracer). The per-event overhead reported for *Acme* is higher as the number of events is lower, reducing the amortisation of *Rehype*'s base (i.e. setup) overhead.

We evaluate two key metrics of tracer overhead: mean per-event overhead and total overhead. Mean per-event overhead is calculated as the total overhead divided by the number of events in the trace-log. That is, the per-event overhead, p, for an execution is given by:

$$p_e = \frac{d_e - b_{\text{avg}}}{c_e} \tag{3.1}$$

where d_e is the duration of the execution e, b_{avg} is the average duration of the base program (program without tracing), and c_e is the number of events in the resulting trace-log. This method of calculating the per-event overhead means that it includes all forms of overhead, including base overhead unaffected by the number of events traced, as such the per-event overhead is an overestimate of the actual average individual cost of an event.

Per-event overhead

To evaluate the per-event overhead I perform 100 executions of the *Acme* test case and Dacapo Lusearch benchmark (using the trace-config that records all Java functions), with tracing, and average the results. Table 3.3 contains the mean per-event overhead, total overhead, and number of events traced for both the *Acme* and Dacapo evaluations. The *Acme* execution captures $\sim 750,000$ events and has a mean per-event overhead of ~ 1500 nanoseconds. The Dacapo execution captures $\sim 630,000,000$ events and has a mean **per-event overhead of ~ 100 nanoseconds**. The events do not take longer to trace in the *Acme* execution, rather the base overhead (i.e. initial setup of the tracer) is simply amortised across fewer events. The Dacapo mean per-event overhead is more accurate to the true per-event cost as the base overhead is amortised across more events.

To put these numbers in a more intuitive, though informal, context; given the per-event overhead calculated here, *Quilt* can trace up to ten million events in one second of user-time (based on ~ 100 nanoseconds per event). Of course, this will not necessarily correlate to one second of wall-clock overhead. However, as a rough rule-of-thumb based on these numbers, if *Quilt* traces less than one million events per-second its overhead should be less than 100% (user-time overhead). In most cases, *Quilt* will not need to trace close to one million events per second, if properly configured to capture only the relevant events. Naturally these informal calculations are dependent upon the execution environment.

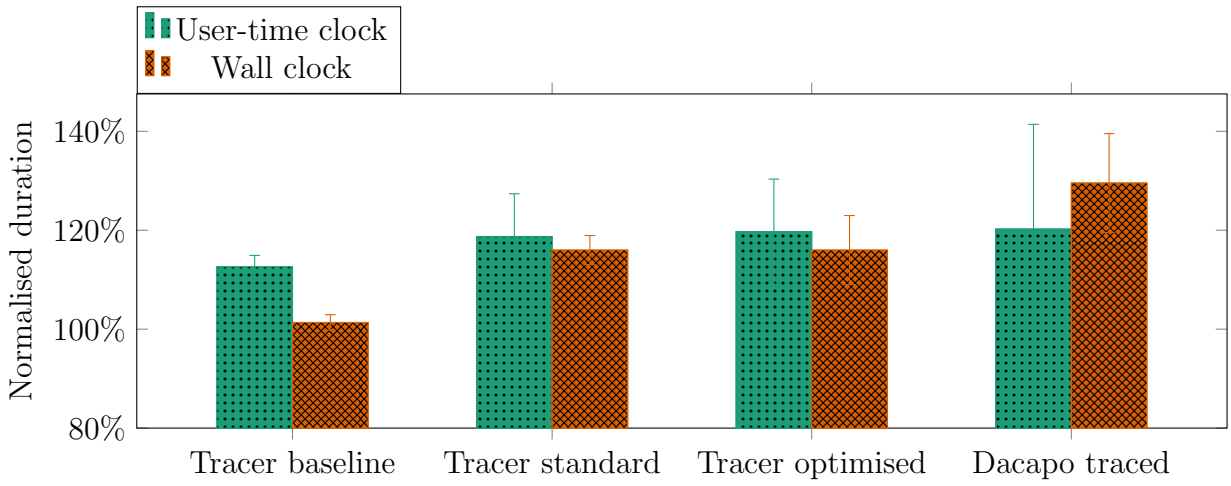


Figure 3.8: The average user-time and wall clock execution durations of *Acme* with various trace configurations and Dacapo Lusearch benchmark with a single trace configuration, averaged across 100 executions, normalised to the execution duration of the program untraced (user-time is normalised to untraced user-time and wall-clock to untraced wall-clock). The key takeaway is that the overhead falls in the target 1 – 10% overhead magnitude range, noted in Section 3.1.1. The error bars represent the standard deviation across the 100 executions.

Total overhead

To evaluate the total overhead incurred by *Quilt*, we trace *Acme* using three different tracer configurations, a *baseline*, *standard*, and *optimised*, and Dacapo's Lusearch benchmark with one configuration. The *Acme standard* configuration is the configuration used in the previous experiment and is the configuration used in evaluations of *Rehype*'s analysis (Chapter 4). The *baseline* configuration captures only the core concurrency events (e.g. task lifetime and thread pool events) and is useful for understanding the base overhead incurred by *Quilt* during setup, as opposed to the overhead incurred by tracing many events. The *optimised* configuration is similar to the standard configuration, except it traces fewer events – still enough for *Rehype* to perform its analysis, but excluding some high frequency, uninformative, events. The trace configuration used for the Dacapo benchmark captures most Lucene functions (the relevant part of the benchmark), excluding a selection of uninformative high frequency functions (e.g. `readByte` functions which get called at a very high rate, but do not provide any relevant information for analysis).

In this experiment we perform 100 executions of Dacapo and *Acme* with each configuration and then average the user-time and wall-clock execution durations for each configuration. Fig. 3.8 illustrates the overhead based on the user-time and wall-clock execution durations for each configuration normalised to their respective untraced execution durations. The base overhead (incurred during *Quilt*'s initialisation) is dominated by the run-time instrumentation (incurred as a Java Agent is used to instrument Java programs). Other components of the base overhead include trace client and buffer-server initialisation, and final flushing of the buffers at the end of the execution. This base overhead is mostly captured in user-time overhead as it is performed on a background thread, this is why the baseline *Acme* configuration has a larger difference between user-time and wall-clock overhead than the other configurations. The other configurations trace more events, which

incurs overhead on the program's working threads, thereby slowing down the execution and hence affecting wall-clock overhead more. The low per-event overhead can be seen in the difference between the standard and optimised configurations. The optimised *Acme* configuration captures $\sim 125,000$ events compared to $\sim 750,000$ captured by the standard configuration, yet this does not significantly change the overhead.

As seen in Fig. 3.8, *Quilt*'s overhead does fall in the 1% – 10% order of magnitude range, given a sensible tracing configuration, as targeted in Section 3.1.1. This places it in a similar range as other existing tracers and profilers. Importantly, the intention of *Quilt* is not to achieve lower overhead that other tracers, but rather to achieve low overhead while tracing the events necessary for the analysis that interests us in this thesis (i.e. analysis performed by *Rehype*).

3.6 Related work

Dynamic analysis is based on data representing the behaviour or state of a program, or underlying hardware, during execution of the program. There are various types of data used for dynamic analysis, approaches to capturing these data, and uses for these data. *Quilt* captures user-space (application) execution traces by instrumenting application code. Other data types include hardware counters (such as CPU and memory usage), stack trace snapshots, program branch tracing, data flow tracing, system call traces, instruction traces, key event logs, and communication traces. Approaches to capturing these data during execution include instrumentation based tracing (including instrumentation of source code, intermediate representations, and binary), system probes, sampling, and framework tracing. Dynamic analyses – uses of these data – include automatic performance analysis, performance profiling for manual performance tuning, automatic bug detection, execution control and replay tools, post-hoc error analysis and reproduction, and program usage analytics.

Though there are three key dimensions, data type, capture approach, and usage, their interactions (dynamic analysis methods) are more prescriptive (e.g. execution replay methods will not use user space sampling as the data is insufficient). As such, I discuss the related work in five categories: execution tracing, sampling, data flow, execution control, and, more broadly, tools that treat overhead as a first-class problem.

Execution tracing occurs continuously and logs all events that are traced, for user-space tracing this generally means function invocations or specific significant abstract execution events (e.g. event-bus communication events). Execution tracing also encompasses instruction tracing (e.g. Intel PIN [40]), kernel tracing (syscall tracing, e.g. Kprobes [50]), application tracing (e.g. Zhang et al. [126]), distributed systems tracing (e.g. Briand et al. [11]), and probe based tracing that spans multiple levels (e.g. DTrace [31]).

Zhang et al.'s Panappticon [126] is a useful contrast to *Quilt* within the application tracing space. Panappticon traces concurrency events in Android applications for post-hoc analysis of *user transactions*. They define user transactions as the operations that occur in an app due to some user interaction, such as tapping a button, and end when some final result is presented to the user. They trace processing across threads at the application, system, and kernel levels and analyse resulting traces for adverse performance effects (e.g. identifying background work that should be deferred until after the user transaction work). Zhang et al. perform light weight tracing to generate minimal data as the tracing

occurs on device and is then transmitted over a network to a processing server. While Panappticon and *Quilt* both perform concurrency tracing their differences are illustrative of variations in data and usage. *Quilt* constrains its tracing to application tracing while also tracing function invocations to enable realistic estimation of traces. Whereas, Panappticon traces span system and kernel events as well, but are constrained to concurrency events that enable it to calculate exact user transaction durations. In both cases exact series of events are important, but the types of events in the traces differ.

Instruction and kernel tracing methods are useful for low-level, in many cases manual, program analysis. Instruction tracing requires execution control (as Intel PIN [40] does) or compile-time instrumentation. Kernel tracing, on the other hand, is commonly implemented using probe-based approaches such as Kprobes [50] and DTrace [31]. Probes are used as, in most cases, the operating system is already compiled (whereas compile-time instrumentation works for most software that a developer is compiling) and they have support for probes given it is a common tool. Kernel system call (*syscall*) tracing can be valuable when identifying causes of significant performance degradation. Specifically, a program's interaction with the underlying operating system, as opposed to user-space program logic, is an indicator of the amount of memory, disk, and network I/O occurring. Thus methods that take the kernel tracing approach can identify key performance impacts while maintaining low tracing overhead as the volume of events can be significantly reduced compared to tracing user-space function invocations.

Sampling methods (also variously known as performance-, CPU-, and memory-profiling), such as Java VisualVM [45] and Linux `perf` [60], stop the execution of a program at a specified frequency (e.g. every 100 milliseconds) and capture a snapshot of the state of the execution. Application samplers such as Java VisualVM capture data such as execution stack traces, while system samplers such as Linux `perf` capture hardware counters such as CPU cycles and cache misses. Sampling can be performed with lower overhead than tracing and its purpose is quite different. Sampling does not provide a concrete series of events, but rather an approximation of the execution based on the captured snapshots. For example, the duration of a function can be approximated based on consecutive samples it appears in. Moreover, a function's impact on program performance can be approximated by its regularity in samples. These approximations are used to generate a high-level profile of program performance (epitomised by Gregg's Flame Graphs [30], which provide a visualisation of program stack traces and their durations based on samples).

Sampling methods and *Quilt* address different tasks. Sampled data would not be suitable for constructing and reordering trace-logs as the operations and key events are not captured. Conversely, the amount of data captured by *Quilt* is demonstrably unnecessary for generating program performance profiles, such as flame graphs, for manual analysis by developers.

Data flow tracing captures usage and movement of memory throughout a program's execution. Data flow analysis can identify potential concurrency bugs and general programming bugs, such as use-after-free. This type of analysis is particularly powerful from the perspective that a program's behaviour is reflected by its data modifications, rather than actions between modifications. As Nethercote & Mycroft [73] put it, "this captures the *essence* of the program's computation, and ignores uninteresting book-keeping details." Data flow analysis is useful for automatic optimisers where full knowledge of data usage is

important. For example, when providing safety guarantees for components of programs, identifying potential optimisations, or providing insight into a program's operation by summarising the program's effects as its data flow.

Another form of data flow tracing is distributed data flow tracing in distributed systems processing (e.g. Wu [125] traces MPI [20] communications to analyse application performance). This type of trace can be used to optimise HPC applications where efficient data flow between nodes in a cluster is key to overall system performance.

Execution control tools span much of the spectrum between low-level and high-level analysis. For example, interactive debugging tools, such as Valgrind [74], control a program's execution so developers can inspect the state of a program at various points in time. Whereas execution replay methods, such as Pobee et al. [90, 89], use execution control to reproduce behaviour observed in concurrent programs that might otherwise be impossible to exactly reproduce. McDonald & Egan [67] use execution replay of C programs in education to help students understand program operation.

Overhead as a first-class problem Mashtizadeh et al. [66] present Castor (a record/-replay system) and Payer et al. [84] present `memTrace` (a memory tracing system), both are examples of systems that treat **overhead as a first-class problem**. Both leverage OS and hardware acceleration to achieve low overhead. For example, Payer et al. trace 32-bit code in 64-bit architectures, allowing `memTrace` to use an additional 2^{32} memory addresses beyond those that are used by the target program, as well as the extra registers available in x86_64 (over x86), meaning that the tracer and program operate on disjoint registers and memory regions. Castor, along with numerous hardware acceleration techniques, achieves low-overhead by being selective in what it records. Specifically, they only record sources of non-determinism (such as system calls). This is a key difference between Castor and *Quilt*; *Quilt* must record even deterministic events (e.g. function entry and exit) to capture enough performance information to accurately estimate the effect of changes and enough program information to identify sensible source-level changes.

Overhead of tools

Finally, it is worth considering the various amounts of overhead different tools incur and where *Quilt* sits on this spectrum.

Dynamic capture tools vary in the patterns and volumes of overhead incurred. At one end are profiling tools (e.g. `perf` [60] and Flight Recorder [44]), which perform low-overhead periodic sampling to provide a rough snapshot of program performance behaviour; incurring small amounts of periodic overhead. At the other end are execution control systems (e.g. PIN [40]) that run programs in a managed environment and intercept CPU instructions; incurring large amounts of overhead constantly, though recording more detailed events to generate traces approximating "full traces". In between there are: 1) kernel instrumentation tools (e.g. DTrace [31] and BPFs [29]) which insert probes in the Linux kernel to trace kernel events; 2) high-level tracing tools (e.g. Ruby Tracer [98]) which incur greater overhead to provide a simpler developer interface; and 3) dynamic instrumentation tools (e.g. Valgrind [74]) that incur high overhead (nullgrind, a minimal Valgrind configuration, has been reported [84] to incur more than 5x overhead on average) but provide a fine-grained tool for analysis of program behaviour, similar to execution

control systems. In terms of overhead, *Quilt* exists between kernel instrumentation tools and high-level tracing tools, it traces more information than kernel tools, but is significantly more optimised than high-level tracers.

A key enabler of this optimisation is *Quilt*'s relatively constrained instrumentation options. Where other tools, such as DTrace and eBPF, provide generic instrumentation aspect-oriented-programming, *Quilt* only supports instrumenting specific aspects and provides no logic based instrumentation definition (such as a aspect API), instead using a relatively plain (fast to parse and interpret) configuration specification. This optimisation is one benefit of tight coupling between tracing and analysis, the tracer can be optimised to capture only the data for the specific analysis, even though such data may be insufficient for other analyses.

3.7 Conclusion

In this chapter I have introduced a low-overhead tracer that acts as the basis for the rest of the thesis' work. The tracer takes in a *trace-config*, instruments Java programs at runtime, and generates a *trace-log* which is used by the analyser in the next chapter to identify possible performance improvements. A key aim of the tracer design is minimal overhead as tracing introduces an observer effect that can affect the validity of subsequent analysis if overhead is too high (see Chapter 6).

To achieve low-overhead tracing the tracer uses a novel lockless buffer exchange protocol, ensuring no blocking operations are performed on the target program's worker threads. This protocol uses a client-server architecture in which there is one client for each thread, which captures all events on that thread, and one overall server that writes events to disk on a background thread. Furthermore, the tracer trades data precision for performance by generating sparse traces and capturing function entry and exit events instead of call-site events. This trace sparsity increases the complexity of subsequent analysis tools (as they have to derive details from sparse information), but is essential for low-overhead tracing.

The tracer introduces wall-clock duration overhead in the order of magnitude range of 1 – 10%, which is similar to existing tracers and profilers. The amount of overhead is primarily driven by the number of events captured. Each event incurs overhead in the order of hundreds of nanoseconds, on average.

Chapter 4

Identifying Concurrency Improvements with Dynamic Analysis

Having captured a *trace-log*, the next step is to analyse it and identify changes that can improve program performance. In this chapter I discuss an approach to analysis that involves estimating effects of changes to identify improvements, and describe *Rehype*, an analyser that uses this approach to identify performance improvements based on reducing concurrency. This analysis is performed purely in the dynamic domain, it does not directly deal with statically implementing the identified changes, though static viability is considered in the analysis. Instead, *Rehype* generates *improvement specifications* which provide the dynamic domain information necessary to implement the static change. The actual patch generation is discussed in Chapter 5.

Narrative instalment

In our imaginary scenario (Section 1.2.2), Banjo has executed an artificial workload on Paterson and captured a *trace-log* with *Quilt*. Now, using the trace-log, Banjo must identify which tasks are causing threads to spend significant amounts of time sleeping. Banjo does this by running *Rehype* on the trace-log, which will generate a list of potential changes and estimations of their effects on performance (both thread usage and wall-clock execution duration). Banjo will then be able to select the changes with the greatest positive impact and implement them as localised incremental changes to the source-code.

4.1 Introduction

In this chapter I address inefficient use of task-based concurrency within existing Java programs that use thread pools to schedule task execution. Task-based concurrency (Section 2.4) simplifies parallelising programs as it is conceptually and logistically straightforward. It is widely used in industrial software. Despite the increasing popularity of reactive programming, task-based concurrency remains a prevalent approach to concurrency within Java 8 (and beyond). Moreover, over the past decade numerous languages (including versions of Rust, C#, Dart, Python, Hack, JavaScript, Scala, and C++) have added support for the `async`/`await` concurrency pattern – in some cases this pattern is

the core concurrency approach for the language. The `async/await` pattern is task-based concurrency, indeed in many languages and runtimes it is a syntactically sugared version of futures.

Inefficient task patterns are easy to fall into and regularly occur in industrial software. A common inefficient pattern is wait-limited tasks. Wait-limited tasks do little computation and instead spend the majority of their execution waiting on other tasks to finish. This is inefficient because a waiting task consumes system resources (e.g. a Java thread) without progressing the program's computation. In a server handling multiple requests, this performance loss manifests as a reduction in *throughput*; it often has no visible effect on either wall-clock or CPU time taken for a solitary request. The cause is clogging of system-wide thread pools. For example, consider a thread pool of eight worker threads with four physical CPU cores; if seven threads are waiting for another task to complete, while only one thread is doing work, then only one core is being used, thus achieving a quarter of the potential performance. The seven waiting threads remain idle, consuming system resources (memory), even while other tasks may be queued in the thread pool, waiting to be executed. In this scenario, refactoring wait-limited tasks can improve performance (for metrics of throughput and wall-clock time, but not classically-accounted CPU time) by freeing threads for other tasks. While these inefficiencies may be resolved by using a different concurrency model (e.g. reactive programming), such refactoring is impractical for large programs, and may introduce other issues. Incremental refactoring reduces risk, while prioritising refactorings improves return on investment – performance benefit per unit developer time.

In many cases there are relatively low-cost solutions (i.e. minimal, localised, source-code improvements) to these inefficient patterns. For example, wait-limited tasks can be either inlined or replaced with combinators (e.g. `CompletableFuture` in Java 8). The two primary inhibitors to improving concurrency use are the difficulties in identifying inefficient tasks and prioritising improvements. Identifying inefficient concurrency patterns statically is difficult as, by their nature, concurrency and performance patterns emerge at runtime. Prioritising improvements is important in industrial contexts where there are limited developer resources and substantial amounts of source code.

In this chapter I introduce and evaluate *Rehype*, a system that performs automatic analysis on runtime execution traces (trace-logs generated by *Quilt*), identifies instances of inefficient concurrency patterns, and suggests potential localised source-code improvements. It estimates the effect of each suggested improvement on the program's performance as a set of metrics, *without re-executing the program.* A developer can then sort and select suggested improvements, based on the metrics, to optimise for their desired performance attributes. *Rehype* produces *improvement specifications* which define how to implement an improvement; it does not generate source-code patches.

While *Rehype* operates solely on dynamic data, in the form of *trace-logs*, I give a static code example of an improvement (Fig. 4.5) to provide intuition (noting that the absence of branching and looping in the example hides various issues). Importantly, *Rehype* aims to suggest only *statically implementable* improvements. If multiple tasks are spawned with identical dynamic context (functions active on the stack) – perhaps due to a source-code loop construct – then they must be treated consistently by the *analyser* (e.g. they must all be inlined, or none of them). I accordingly distinguish the idea of *optimisation*, such as *Rehype*'s *convert-to-inline*, from its instances (here called *improvements*) which are concrete modifications of *trace-logs*. For the case above, the improvement might be expressed as

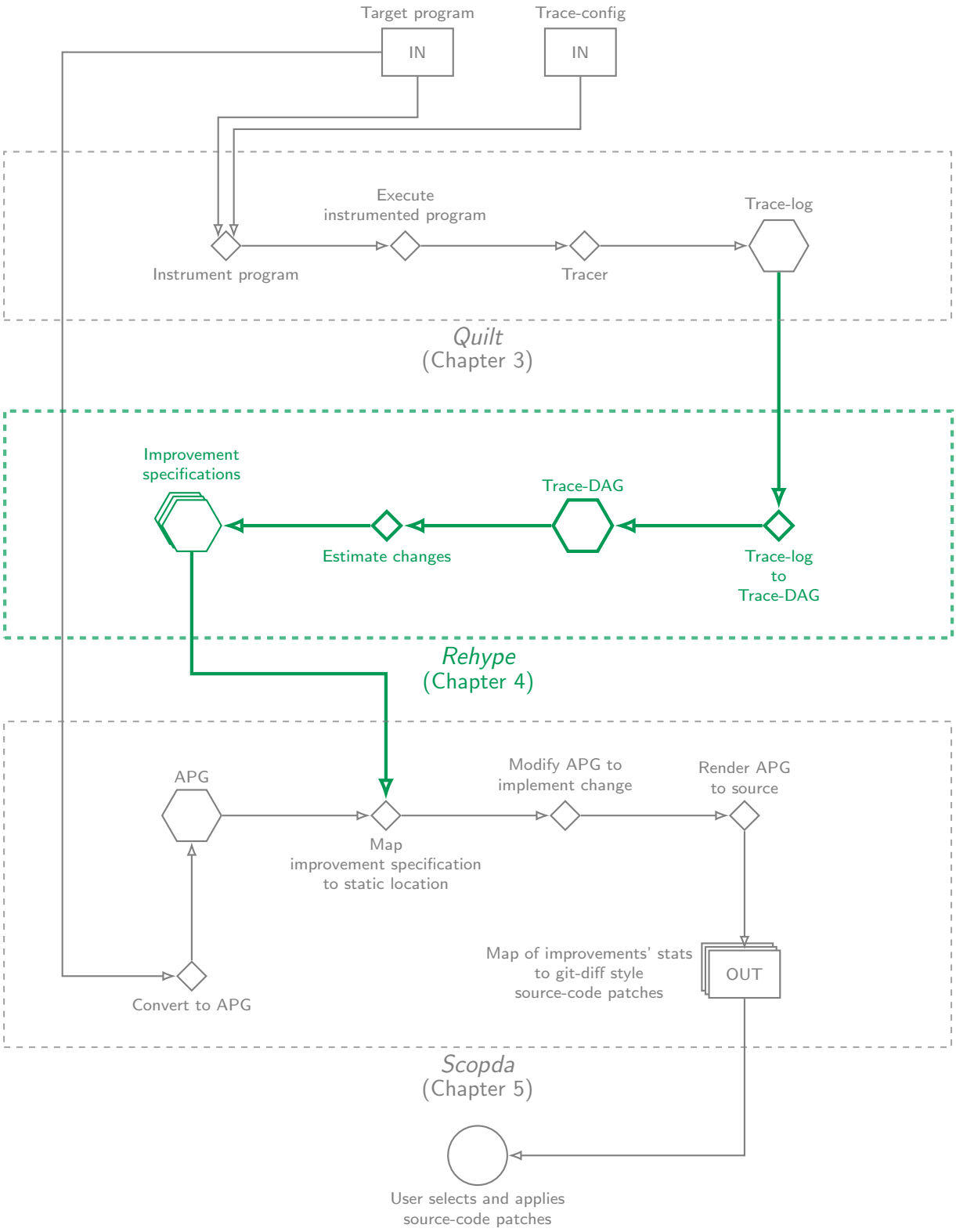


Figure 4.1: High-level overview of where *Rehype* exists within *DTRSO*. *Rehype* takes in the trace-log generated by *Quilt*, converts it to a *trace-DAG* for processing, and estimates a set of changes as *improvement specifications* which are then used by *Scopda* (Chapter 5).

"inline every task spawned at dynamic context X".

A key idea is that *Rehype* identifies improvements by estimating the performance effects of implementing them. This estimation is performed by manipulating a *trace-log*. It occurs offline and does not require re-executing the program. Using an estimation-based approach enables *Rehype* to identify improvements in programs of varying complexity and for a variety of patterns, as it does not *detect* specific patterns, but *tests* specific improvements. This approach is well suited to concurrency performance optimisation as it can test the effect of implementing multiple interfering improvements. *Rehype* can accurately estimate the effects of changes to concurrency (see Section 4.3.2) as the *trace-log* contains task life-cycle events that define the inter-thread concurrent dependencies, allowing *Rehype* to account for the concurrent behaviour.

Implementing a combination of improvements suggested by *Rehype* for *Acme* (Section 2.6.1), a large real-world server program, more than doubled its request processing throughput (Section 4.3). While there is a lot of hype around increasing concurrency, *Rehype*'s results show that reducing concurrency can significantly improve throughput in resource constrained programs, such as application back-end servers.

Structure of *Rehype*

Rehype takes a *trace log*[1] generated by *Quilt* and produces a list of suggested *improvement specifications* for the program along with their predicted performance effects. The *trace-log* events record the behaviour of the program via function call events and the inter-thread concurrent dependencies via concurrent construct events (e.g. task lifetime events for task-based concurrency). Inter-thread concurrent dependencies occur when one *thread* must wait for another *thread* to reach some point (e.g. completing a Java `FutureTask`) before continuing. These events enable *Rehype* to re-order traces to estimate the performance effect of different potential changes by moving events while maintaining the inter-thread dependencies.

Target concurrency model refresh

Recall from Section 2.4, in this thesis we focus on a task-based concurrency model using thread-pool scheduling (discussion on other related models, such as Fork-Join, is in Section 2.4.3). As a quick refresher, *tasks* are concurrent constructs that enable scheduling the execution of a function (here called the *task-body function*) with some parameters. *Rehype* specifically analyses a *future*-based model of tasks, whereby spawning a task generates a *future* representing the return value of the task. When the task's result is retrieved from the future, the call blocks until the task has finished (providing synchronisation). *Rehype* assumes[2] that concurrent communication occurs solely at task

[1] In my implementation, the *trace log* is first pre-processed by a separate *trace processor*, but this is not conceptually important.

[2] Unfortunately, it is not possible to determine automatically whether a Java program adheres strictly to this concurrency model. While it is possible to check instances of the model being broken (e.g. clear usage of shared memory for coordination), proving the inverse (that the model is adhered to) is undecidable due to the aliasing problem (objects may be aliased and used between multiple tasks in non-obvious ways). However, new type systems, such as Rust's ownership system, may go some of the way towards enabling better model adherence checking.

spawn and on retrieving future-based results[3]. Thus, in essence, tasks present a pure-functional interface with the exception of operations that do not affect the result of the program, such as logging.

Intuitively, the mental model is of tasks corresponding to pure functions that are called and eventually return a single result (via a future). Specifically a task's logical behaviour (modulo memory locations, etc) is determined by its parameters. In practice, tasks will rarely be true pure functions, they may depend on, and mutate, some external state (such as a database). However, the key is that such state dependence is not relied upon for inter-task communication (i.e. a task does not rely on another, concurrently executing, task having edited that state).

Thread pools contain a fixed number of worker *threads* and a queue of *tasks* to execute. We assume that all *threads* in a *thread pool* are solely used for executing *tasks*. In this thesis, *threads* refer to the source-code-level logical threads. Although threads may be executed on various physical CPU cores, depending on the operating system scheduler, this is below the level of detail considered in this thesis.

Threads exist in one of three states: 1) *active* when executing a task; 2) *waiting* when asleep waiting for the result of another task; and 3) *unoccupied* when waiting to be assigned a task by the thread pool. To maximise performance, programs should have most threads *active* most of the time. While *waiting*, a thread is consuming system resources but not actively progressing its current *task*. An *unoccupied* thread is available for use and consumes fewer system resources.

4.2 Method

Rehype takes a *trace-log* and generates a list of suggested improvements and their estimated affect on throughput; Section 4.2.3 details the three *optimisations* we consider. For the purpose of this chapter, throughput is represented as a combination of execution duration and per-nanosecond thread usage – number of threads in each of the three work states, *active*, *waiting*, and *unoccupied*. *Rehype* individually estimates the effect of each possible *optimisation* for every task and then estimates the combined effect of sets of compatible *improvements*.

Rehype estimates the effects of potential changes, rather than attempting to detect potentially inefficient concurrency patterns. In some instances a change may improve throughput, in others it may degrade throughput. *Rehype*'s estimation algorithm predicts whether a change improves or degrades throughput. All three optimisations *Rehype* considers relate to the concurrency of execution of tasks. Importantly, these optimisations do not affect the program's functionality, provided the program adheres to the "all concurrent communication occurs through tasks (i.e. futures)" model explained in Section 2.4.

Rehype individually estimates the effect of each possible optimisation for every task and then estimates the combined effect of sets of compatible changes (instances of optimisations).

[3]As an indicator, of the 938 196 Java files that import `ExecutorService` on GitHub (see introduction), 216 714 also contain the `synchronized` keyword, suggesting, though not confirming, that they do not adhere to this pure task model.

```
static void main(String[] args) {
  example();
  otherExample();
}

static void example() {
  spawn();
}


static void otherExample() {
  spawn();
}

static void spawn() {
  // Assume executorService is available.
  executorService.submit(/* ... */);
}
```

(a) Example program that would generate two tasks with different, but similar, *dynamic contexts*.

```
example() task: [main, example, spawn, ExecutorService.submit, FutureTask.<init>]
otherExample() task: [main, otherExample, spawn, ExecutorService.submit, FutureTask.<init>]
```

(b) The spawn dynamic-contexts of the tasks that would be generated by executing the code in Fig. 4.2a.

Figure 4.2: An example program and the spawn dynamic-contexts that would be generated for its tasks.

4.2.1 Dynamic context

Tasks are defined by the *dynamic contexts* of their spawn and execution events. The dynamic context is defined by the function events surrounding the task event. For example, the dynamic context for a spawn event is defined by the active call-stack when the event occurred. The active call-stack is calculated as the `FnStart` events before the `TkCtor` event, that do not have matching `FnEnd` events. However, the dynamic context for the `TkExeStart` event is defined as the function invocation tree occurring after the event (i.e. what code was executed by the task). Of course, this means that the contexts have different structures, the spawn context is a list while the execution context is a tree.

For example, in Fig. 4.2a two tasks will be spawned, one through the `example` function and the other through the `otherExample` function. Their spawn dynamic contexts would resemble the lists shown in Fig. 4.2b.

Whereas, the function in Fig. 4.3a would generate two tasks with identical spawn dynamic contexts, but different execution dynamic contexts. Their execution dynamic contexts would resemble Fig. 4.3b.

4.2.2 Task groups

Task groups are used to ensure change estimations are statically valid. Task groups are constructed to represent all tasks created by a particular piece of code, such that changing

```
static void start() {
  executorService.submit(() -> example());
  executorService.submit(() -> otherExample());
}
```

(a) Example program that would generate two tasks with identical spawn dynamic contexts, but different execution dynamic contexts.

```
example() task: {Future.run(): { lambda1: { example(): { ... }}}}
otherExample() task: {Future.run(): { lambda2: { otherExample(): { ... }}}}
```

(b) The execution dynamic-contexts of the tasks that would be generated by executing the code in Fig. 4.3a. The tasks' spawn dynamic-contexts would be the same. Recall that these are trees.

Figure 4.3: An example program and the execution dynamic-contexts that would be generated if it were executed.

```
[spawn, ExecutorService.submit, FutureTask.<init>]
```

Figure 4.4: A *partial* spawn dynamic-context that the tasks from Fig. 4.2a can be group by.

the code to implement an optimisation would affect all of the tasks within the group. This means that, when estimating a change, the optimisation must be applied to all tasks within a task group, otherwise it would be estimating a change that could not be statically made. Of course, a task group could contain a single task if that piece of code were run just once.

Task groups are constructed based on shared dynamic contexts, including partial dynamic contexts. For example, the two tasks in Fig. 4.3a could be grouped as they share the same spawn dynamic context. Also, the two tasks from Fig. 4.2a can be grouped as, though they have different *complete* spawn dynamic contexts, they share a *partial* spawn dynamic context, shown in Fig. 4.4. Similarly, if the `example` function from Fig. 4.2a were executed in a loop, it would generate multiple tasks that share a complete dynamic context and would form a task group (though they are executed at different times, they would have the same dynamic contexts).

The full set of task groups for the tasks in a trace is generated by recursively separating task groups, creating a tree of task groups. The root of each tree is defined by the minimal partial dynamic-context such that it captures the most tasks it can (there may be multiple trees if some tasks do not share any partial dynamic context). Then, for each task group in the tree, split it into two task groups by increasing the specificity of the parent task group's dynamic context. An example of increasing specificity is increasing the length of the spawn dynamic-context until the spawn dynamic-contexts differ. How to split a task group is determined by which increase in specificity (as, normally, multiple will be possible) best separates the contained tasks (defined as the most even split of tasks). Tasks may be

contained by multiple task groups (i.e. task groups are not necessarily disjoint), as there may be multiple ways to group tasks.

4.2.3 Optimisations

Rehype considers three optimisations: *convert-to-inline*, *hoist-branch*, and *convert-to-combinator*. The optimisations convert some concurrently executed tasks in a program to execute immediately at spawn (i.e. inline). The *optimisations* differ on how the source change would be implemented and thus which tasks are changed. Selectively removing concurrency can reduce the number of threads *waiting*, improving performance by freeing them to be used by other tasks. The optimisations differ on how the source code would be changed and thus which tasks are affected.

Convert-to-inline

In the *convert-to-inline* optimisation, *Rehype* estimates the effect of executing the *task* immediately on construction (on its constructing thread), instead of spawning a task to be executed on a separate *thread*. Note that the thread-pool model implies that the thread which executed the *task* necessarily differs from the thread which constructed it[4].

For example, in Fig. 4.5, pre-change `queryAge()` invokes `queryDatabase()` and immediately waits on the returned `Future`. Thus, for the duration of the task two threads are occupied, one purely for waiting. The *convert-to-inline* optimisation improves performance by reducing the number of occupied threads without affecting execution duration (or even reducing it depending on the task scheduling overhead). After the change the top-level task function, `queryDatabase()`, becomes synchronous[5], improving throughput (and thus performance) by reducing thread usage without affecting execution duration.

A subtlety is that multiple tasks may be spawned at the same dynamic context and would thus all be affected by the same code change. To account for this, the *convert-to-inline* optimisation is estimated for the set of tasks spawned at the same dynamic context.

Hoist-branch

The *convert-to-inline* optimisation can be somewhat of a blunt instrument given its all-or-nothing approach. The *hoist-branch* optimisation extends the *convert-to-inline* optimisation by identifying a subset of the tasks that can be inlined without affecting the other tasks. For example, when one subset of tasks performs extensive work, suitable for a task, and another subset returns quickly, it can be beneficial to inline only the latter subset. In the simple form, this difference in execution may be caused by an `if-then-else` (or `switch`) statement. The optimisation can be practically implemented by *hoisting* (moving) the `if-then-else` statement out of the task execution and to the task spawn code. Such cases of divergent executions can be identified in program traces based on the execution

[4]Tasks that are submitted to a thread-pool and immediately executed on the submitting thread due to a thread-pool implementation policy (e.g. `ThreadPoolExecutor.CallerRunsPolicy()` [43]) are ignored by *Rehype*.

[5]In this example we assume `queryDatabase()` is only called by `queryAge()`. In practice the refactor may take the form of a new function, such as `queryDatabaseNoTask()`, so the original method signature remains for all other usages.

```
class AgeQuerier {
  private final ExecutorService executorService;

  // ... constructor creating executor service ...

  private Future<DatabaseRecord> queryDatabase(String id) {
    return executorService.submit(() -> {
      DatabaseRecord queryResult = /* ... perform some database query ... */;
      return queryResult;
    });
  }

  int queryAge(String id) {
    // Usages that immediately wait on the result of the task are simplified.
    DatabaseRecord result = queryDatabase(id).get();
    // Return the age inner value of the record.
    return result.getAge();
  }
}
```

Figure 4.5: Example of the convert-to-inline optimisation applied to an inefficient task. Before the change, `queryAge()` calls and immediately waits for the result of `queryDatabase()`, effectively executing the task synchronously while requiring two threads. The change makes the task synchronous by removing the `executorService.submit()` call and returning the value directly, instead of inside a `Future`.

dynamic contexts of the tasks (whereas *convert-to-inline* is purely based on the task-spawn dynamic context); when there is a hoist-able branching statement, the execution dynamic contexts of the subsets of tasks will differ (i.e. there will be two task groups that share a spawn dynamic context, but have different execution dynamic contexts). The dynamic estimation of inlining a subset of tasks is the same as *convert-to-inline*, hence the optimisation is an extension of *convert-to-inline*. In fact, from a purely dynamic perspective, the optimisation could be called *convert-subset-to-inline*, however, the term *hoist-branch* better matches the practical change a developer would implement.

While there are more complex causes of branching, such as dynamic dispatch, the difference between task execution paths is similar whether due to a simple `if-then-else` or a more complex branching mechanism. As such, *Rehype* treats them similarly, though the static implementation may be more complex.

Rehype's traces do not explicitly identify branching statements, instead the existence of a branching statement is inferred from the difference in task execution dynamic contexts. For example, in Fig. 4.6, pre-change, the existence of the branching statement (`if (cached.isPresent())`) is inferred from multiple tasks sharing the spawn dynamic context `[getValue]` with some including `queryRemoteData()` in their execution dynamic contexts (when the value is not cached), and others not including the events (when the value is cached). As such, *Rehype* outputs the dynamic context information (i.e. task spawn contexts and the various in execution contexts) to identify the branching statement to be hoisted, but cannot explicitly identify the statement (e.g. with a file & line number). As `FnStart`/`FnEnd` only contain a reference to the function invoked, not to the call-site, *Rehype* cannot directly disambiguate between two calls to a function from the same caller function (i.e. the caller-context is limited, see Section 3.3).

Fig. 4.6 is an example of a source change hoisting a branch out of a task execution to improve throughput. A task, `getValue()`, checks a fast local cache and returns the

```
class CachedDatabaseQuerier {
  private final ExecutorService executorService;

  // ... constructor creating executor service ...

  Optional<DatabaseRecord> getCached(String id) {
    // ... check fast local cache
  }

  DatabaseRecord queryRemoteDatabase(String id) {
    DatabaseRecord result = /* query some remote database */;
    return result;
  }

  Future<DatabaseRecord> getValue(String id) {
    Optional<DatabaseRecord> cached = getCached(id);

    // Check if value is cached without using concurrency.
    if (cached.isPresent()) {
      // To keep the API consistent the cached branch returns a future that
      // is immediately resolved.
      CompletableFuture<DatabaseRecord> immediate = new CompletableFuture<>();
      immediate.complete(cached.get());
      return immediate;
    } else {
      return executorService.submit(() -> {
        Optional<DatabaseRecord> cached = getCached(id);

        // Check if value is cached.
        if (cached.isPresent()) {
          return cached.get();
        } else {
          return queryRemoteDatabase(id);
        }
      });
    }
  }
}
```

Figure 4.6: Example of a source change to hoist a branching statement from task execution to stop inefficient occurrences being concurrent. Specifically, in `getValue()`, if the value is cached using concurrency is inefficient, however, if the value is not cached, concurrency is beneficial. This source change hoists the branch statement (`if (cached.isPresent())`) out of execution so concurrency is only used if the value is not cached.

value from the cache if present, otherwise it queries a remote database. When the value is cached the task is fast enough that inlining it would not significantly affect execution duration and would reduce thread usage, improving throughput. However, when the value is not cached the task involves network I/O and so inlining the task would negatively affect throughput. Applying *convert-to-inline* would be insufficient as both cases would be inlined (i.e. the network I/O would be inlined). Instead, the branch can be hoisted so the cache is checked before the task is placed on a thread. Thus only the instances when the task is cached are inlined.

Convert-to-combinator

In Java, concurrency combinators can encode task scheduling and coordination (e.g. passing the result of one task as an argument to another task). Perfect use of combinators can result in sleep-free task concurrency as no thread ever waits for a task to finish (Section 2.4).

```
static Future<DatabaseRecord> parallelSum(ExecutorService es) {
  return es.submit(() -> {
    Future<Integer> first = es.submit(Example::longOperation1);
    Future<Integer> second = es.submit(Example::longOperation2);
    return first.get() + second.get();
  });
  CompletableFuture<Integer> first = CompletableFuture.supplyAsync(
        Example::longOperation1, es);
  CompletableFuture<Integer> second = CompletableFuture.supplyAsync(
        Example::longOperation2, es);
  return second.thenCombine(first, (f, s) -> f + s);
}
```

Figure 4.7: Example of the convert-to-combinator optimisation.

Instead of some code, x, waiting on a thread to use a task's result, a combinator represents x as a *task-body function* to be executed when the task result is ready.

Combinators are not as intuitive or straightforward to use as they require designing the task scheduling explicitly, instead of programming sequentially and relying on futures to determine synchronisation (whereby scheduling is handled by imperative code and threads waiting). Moreover, combinators are not always easy to practically retrofit to existing code.

The *convert-to-combinator* optimisation replaces *coordinator tasks*, tasks that primarily exist to coordinate other tasks, with combinators. This frees a thread to be used by other tasks. The optimisation determines the scheduling of the tasks spawned by the *coordinator task* and encodes it in a combinator (specifically a `CompletableFuture` in Java 8). Fig. 4.7 illustrates the optimisation in a simple parallel summation example.

A subtlety is that *Rehype* does not *detect* coordinator tasks, rather it estimates the effect of converting each applicable task (those that create and wait for another "sub-task") to a combinator. This embodies the key idea of *Rehype*: "identify improvements by estimating the performance effects of implementing them" key idea.

4.2.4 Estimation

To estimate the performance effects of changes, *Rehype*[6] re-orders events from a *trace-log* to produce a new *trace-log* which approximates a *trace-log* (modulo exact nanosecond timings encoded as $\simeq$ below) that would be generated by re-executing the original program after applying the suggested change(s). *Rehype* aims to generate a realistic *trace-log* that is as accurate as possible to the actual log the program would generate.

More concretely, assume some given program p_0 and suggested source-level changes, δ, for it. Now write program p_δ for the resulting program and let $[\![p]\!]$ stand for the *trace-log* obtained by executing program p on some fixed given data. Then *Rehype*'s estimation method is a function β_δ on *trace-logs* such that:

$$\beta_\delta \, [\![p_0]\!] \simeq [\![p_\delta]\!] \tag{4.1}$$

In other words the source-code change δ is reflected as a trace-log change β_δ.

We implement β using a *trace-DAG* (directed acyclic graph) of the *trace-log*. A *trace-DAG* is a structured representation of a *trace-log*. This makes reordering events simpler

[6] It not only analyses but also transforms!

in the structured *trace-DAG* as edits are made by moving edges and the effects of edits on the rest of the trace are automatically propagated to affected events. The high-level process of β_δ is: transform *trace-log* into a *trace-DAG*, apply δ changes to the *trace-DAG*, and transform the *trace-DAG* back into a *trace-log*.

Where a *trace-log* has absolute values per-event, a *trace-DAG* has relative values on edges or inherited via edges. The use of relative values, instead of absolute values, significantly simplifies editing traces. For example, in a *trace-log* each event has an absolute timestamp. By contrast in a *trace-DAG* temporal information is stored as *durations* (relative timings) and used to label edges. (Vertices in a *trace-DAG* have no explicit timing information, but absolute timings of vertices can be calculated from the constraints imposed by time edge durations.)

More formally, a *trace-DAG* is a triple (V, E^Θ, E^τ) where V is a set of vertices (one for each event in the *trace log*) and E^Θ, E^τ are sets of directed dependency edges (respectively *thread edges* and *time edges*). By construction *trace-DAGs* are acyclic. There are two types of edges: *thread edges* and *time edges*. The former are unlabelled and chain together successive events executed by a single thread;[7] the latter are labelled with durations which express a minimum delay between two events. Two vertices connected by a thread edge are always connected by a time edge as well. Though, this condition is not strictly bidirectional, two vertices not connected by a thread edge may still be connected by a time edge. This occurs between specific task life-cycle events associated with the same task. The two edge types in more detail are:

Thread edge (E^Θ): encodes which thread a node occurs on and the order of events in the thread. Given an edge (v, w), the event for w occurs immediately after the event for v and occurs on the same thread. Every vertex has exactly one incoming and one outgoing thread edge, with the exception of the first and last events on a thread.

In diagrams I draw thread edges as a double arc.

Time edge (E^τ): encodes the time delta between vertices with a nanosecond duration label. Given an edge (v, w, d), where d is the nanosecond duration label, the event corresponding to vertex w must happen *at least* d nanoseconds after v. This constraint means that if a *trace-log* is derived from a *trace-DAG* (see Section 4.2.4.4) without the *trace-DAG* being changed, the derived *trace-log* events will be exactly the same as the original *trace-log* that the *trace-DAG* was created from. However, edits made to the *trace-DAG* (see Section 4.2.4.3) may produce graphs in which these constraints need to be treated as inequalities.

Vertices may have any number of incoming and outgoing time edges. If a vertex has multiple incoming time edges, then re-creating its associated *trace-log* timestamp needs to satisfy the duration constraints of all its incoming edges (see Section 4.2.4.4).

Time edges between vertices on different threads enforce inter-thread "happens-before" constraints by using a zero duration edge label. For example, a `TkCtor` event vertex will have outgoing time edges to every other task-event vertex for that task with the exception of transitively redundant time edges.

In diagrams I draw time edges as a single arc with duration d written alongside.

[7] These events may represent actions of distinct tasks, but due to the thread-pool model all events corresponding to a single task occur contiguously on this chain.

Trace-DAGs are transitively irreducible. That is, given edges $v_0 \to v_1$ and $v_1 \to v_2$, no edge $v_0 \to v_2$ exists. Transitive irreducibility is necessary so edits to the graph behave as expected with regards to vertex order. For example, given a graph with five vertices $\{v_0,\ v_1,\ v_2,\ v_3,\ w\}$ and edges $v_0 \to v_1 \to v_2 \to v_3$, an edit to the graph can "move" v_1 and its descendant vertices to a new parent, w, by removing the edge $v_0 \to v_1$ and adding the edge $w \to v_1$. Performing such an edit in a transitively irreducible graph has well defined outcomes (v_1, v_2, and v_3 become descendants of w and not v_0). However, such edits have undefined behaviour if the graph is transitively reducible (e.g. if $v_0 \to v_2$ existed then v_2 and v_3 would be descendants of both v_0 and w). Moreover, trace-DAGs are conceptually transitively irreducible as each vertex only depends on the immediate preceding vertices.

4.2.4.1 Soundness

Rehype's re-ordering of the *trace-log* is sound, given adherence to the task concurrency model described in Section 2.4. That is, the re-ordered *trace-log* reflects the *trace-log* the program would generate given the immediate execution of the *task-body function*. Recall that the concurrency model assumes solely future-based concurrent communication. In particular, tasks intuitively correspond to (externally) pure functions and their behaviour is determined by their parameters. Thus, a task will behave the same whether it is executed immediately (as a direct call to the *task-body function*) or scheduled on a thread-pool; therefore re-ordering of task execution events in a *trace-log* is sound.

4.2.4.2 Trace-DAG construction

The *trace-DAG* is a triple of $(V, E^{\Theta}, E^{\tau})$. Initially the trace-DAG contains one vertex for each event in the *trace-log*. For implementation efficiency some vertices can merged[8]. Merging vertices for efficiency does not change the conceptual formulation or function of the *trace-DAG*. As such the rest of this chapter assumes no merging of vertices.

The edges in a graph are derived from four rules for related events, they are: the first and last events on a thread are connected (with both edges and time edges of duration zero) to special thread start and end vertices; two consecutive events on a thread; `TkCtor` events and all other events associated with the same task; and `TkResSet` events and `TkResGetEnd` events associated with the same task. While explicit thread start and end events are unnecessary in trace-logs (and thus do not exist), it is convenient to insert special thread start and end vertices in the trace-DAG to simplify graph edit operations. Specifically, graph edit rules (Section 4.2.4.3) assume all vertices being edited have at least one preceding and succeeding vertex. The special thread start vertices also initialise the absolute thread id values which are inherited by normal vertices through the thread edges.

Formally, let $L = [\![p]\!]$, the trace-log captured from an execution of the program p with data as before. For any thread θ, write L^{θ} as the list of ordered events occurring on the thread and write L^{θ}_i as the i^{th} event in the thread (ranging from 0 to $n-1$, where n is the number of events in the thread). Let $start(\theta)$ and $end(\theta)$ be the special thread start and end vertices, respectively, for thread θ. Assume a mapping from event to vertex ϕ such that $\phi(e)$ is the vertex for e and its inverse mapping ψ. Write events as data structures per the definition in Fig. 3.2 (Section 3.2) – i.e. for event e, $e.timestamp$ is the nanosecond

[8]We merge vertices whose corresponding events are function events and are contiguous (i.e. the events form a block of code). This is similar to basic blocks in compilers.

timestamp for the event e). Then the edge construction rules are[9]:

Rule 1: Given the first and last events on a thread θ, L_0^θ and L_{n-1}^θ, respectively, add:

$$E^\Theta : (start(\theta),\ \phi(L_0^\theta)) \tag{4.2}$$
$$E^\tau : (start(\theta),\ \phi(L_0^\theta), 0) \tag{4.3}$$
$$E^\Theta : (\phi(L_{n-1}^\theta),\ end(\theta)) \tag{4.4}$$
$$E^\tau : (\phi(L_{n-1}^\theta),\ end(\theta), 0) \tag{4.5}$$

Rule 2: Given two consecutive events on a thread, L_i^θ and L_{i+1}^θ, for the thread θ, add:

$$E^\Theta : (\phi(L_i^\theta),\ \phi(L_{i+1}^\theta)) \tag{4.6}$$
$$E^\tau : (\phi(L_i^\theta),\ \phi(L_{i+1}^\theta),\ L_{i+1}^\theta.timestamp - L_i^\theta.timestamp) \tag{4.7}$$

Rule 3: Given a `TkCtor` event e^{TkCtor}, for every other event associated with the same task $e^{\text{Tk}*}$, add:

$$E^\tau : (\phi(e^{\text{TkCtor}}),\ \phi(e^{\text{Tk}*}),\ 0) \tag{4.8}$$

Two events e, e', are associated with the same task if $e.task\_id = e'.task\_id$.

Rule 4: Given a `TkResSet` event e^{TkResSet}, for every `TkResGetEnd` event associated with the same task $e^{\text{TkResGetEnd}}$, add:

$$E^\tau : (\phi(e^{\text{TkResSet}}),\ \phi(e^{\text{TkResGetEnd}}),\ 0) \tag{4.9}$$

Refer to Table 3.2 in Section 3.2 for the definition of all task life-cycle events.

Recall, the duration labels in rules 3 and 4 are zero as they are concurrent "happens-before" constraints. Practically, such concurrent constraint edges only affect timestamps when a *trace-DAG* is edited such that the destination of the edge would otherwise occur before the origin.

Trace-DAGs must be transitively irreducible, as such we only record immediate dependencies (non-transitive edges), as is common in dependency graph construction. Thread edges are naturally non-transitive as there is only ever at most one incoming and one outgoing thread edge from a vertex. For time-edges transitive irreducibility is an extra constraint applied to rules 3 and 4, i.e. if a path exists between vertices v and w, do not add a new time edge (v, w, d). In my implementation I generate edges based on rules 1 and 2, then rules 3 and 4 where both events are not from the same thread (i.e. already transitively connected), and finally perform transitive reduction to remove transitively redundant inter-thread edges. Though in many cases there are no inter-thread transitively reducible edges, they can occur in some task usage patterns. Performing transitive reduction as a final step is more efficient than checking, on every edge insertion, transitive connectedness between vertices not on the same thread.

[9]While the notation $(v,\ w) \in E$ is more common for rule-based construction of sets, I use the $E : (v,\ w)$ notation as it is more consistent with the graph grammar used for editing graphs in Section 4.2.4.3.

4.2.4.3 Optimisation graph-edits

Optimisations are estimated by editing the *trace-DAG*. All three optimisations described, *convert-to-inline*, *convert-to-combinator*, and *hoist-branch*, are based on the same core graph edit: moving the execution events of a task to the position of its `TkCtor` event. The execution events for a task are all events on the task's execution thread between `TkExeStart` and `TkResSet`, inclusive (i.e. all events in the thread edge path between `TkExeStart` and `TkResSet`). The execution events replace the `TkCtor` vertex.

Convert-to-inline applies this edit to the given task.

Convert-to-combinator applies it to all *immediate owned-sub-tasks* of the given task. *Immediate owned-sub-tasks* are tasks that are created and used in the same task (i.e. `TkCtor` and `TkResGetStart` events both occur in the task's execution).

To express this more formally, first define $v \overset{\Theta^*}{\Longrightarrow} v'$ to mean that there is a thread-edge path from v to v' (i.e. $\overset{\Theta^*}{\Longrightarrow}$ is the transitive closure of thread edges seen as a relation). Also, we extend the v^{TkCtor} notation so that $v^{\text{TkCtor}(x)}$ means that vertex v corresponds to a `TkCtor` event whose *task_id* is x. Then task y is an *immediate owned-sub-task* of task x if there are vertices $v_1, \ldots v_4$ such that

$$v_1^{\text{TkExeStart}(x)} \overset{\Theta^*}{\Longrightarrow} v_2^{\text{TkCtor}(y)} \overset{\Theta^*}{\Longrightarrow} v_3^{\text{TkResGetStart}(y)} \overset{\Theta^*}{\Longrightarrow} v_4^{\text{TkResSet}(x)}$$

Hoist-branch applies the edit to a specific subset of tasks spawned with a common dynamic context, where the subset share an execution dynamic context.

As the scope of a task's execution is the vertices between the `TkExeStart` vertex and the `TkResSet` vertex, moving the execution of a task is implemented by redirecting edges between task event vertices. The edges terminating at the `TkExeStart` vertex and originating from the `TkResSet` vertex are stitched together, removing the task execution from its current thread. Edges terminating at the `TkCtor` vertex are directed to the `TkExeStart` vertex and edges originating at the `TkCtor` vertex are moved to originate from the `TkResSet` vertex, inserting the task execution into the task's construction thread.

Formally, given a task to edit and its associated vertices v^{TkCtor}, $v^{\text{TkExeStart}}$, and v^{TkResSet}. Write the replacement of edge(s) X with edge y as $X \mapsto y$. Using this graph notation, the graph edit for inlining a task's execution is:

Stitch execution thread, removing task execution from the thread:

$$E^{\Theta} : (s,\ v^{\text{TkExeStart}}),\ (v^{\text{TkResSet}},\ w) \mapsto (s,\ w) \tag{4.10}$$

$$E^{\tau} : (s,\ v^{\text{TkExeStart}},\ d_1),\ (v^{\text{TkResSet}},\ w,\ d_2) \mapsto (s,\ w,\ d_1 + d_2) \tag{4.11}$$

Insert task execution into construction thread:

$$\begin{aligned} E^{\Theta} :\ & (s,\ v^{\text{TkCtor}}) \mapsto (s,\ v^{\text{TkExeStart}}) \\ & (v^{\text{TkCtor}},\ w) \mapsto (v^{\text{TkResSet}},\ w) \end{aligned} \tag{4.12}$$

$$\begin{aligned} E^{\tau} :\ & (s,\ v^{\text{TkCtor}},\ d) \mapsto (s,\ v^{\text{TkExeStart}},\ d) \\ & (v^{\text{TkCtor}},\ w,\ d) \mapsto (v^{\text{TkResSet}},\ w,\ d) \end{aligned} \tag{4.13}$$

Note, this edit removes v^{TkCtor} from the graph as it is redundant post edit. This algorithm is illustrated in Fig. 4.8.

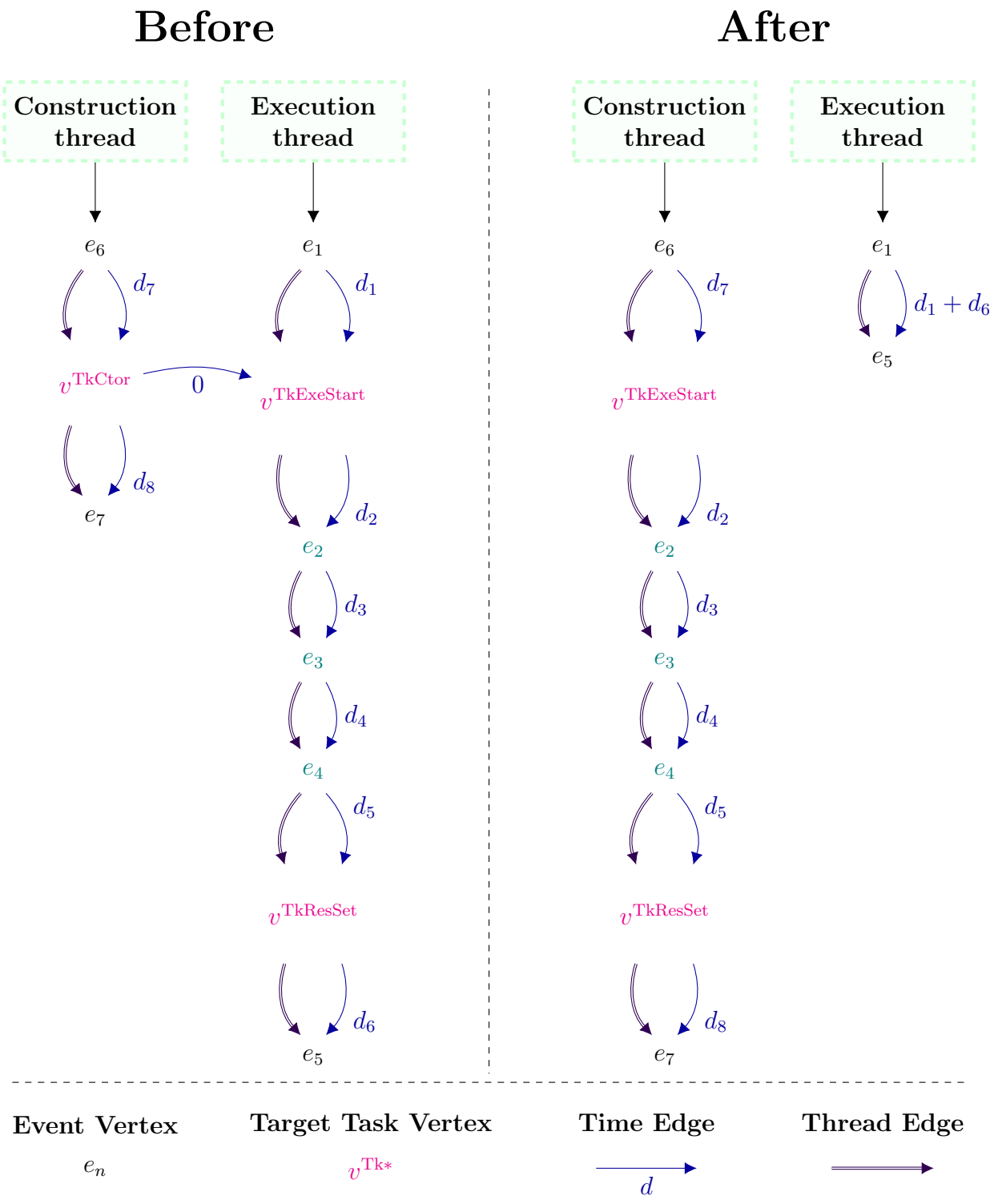


Figure 4.8: Graph edit operation to inline task execution. See text for description and formalisation (Eq. 4.10-4.13) of edit.

4.2.4.4 Trace-DAG to Trace-log derivation

To derive a *trace-log* from a *trace-DAG*, events are created for most vertices in the graph and written, in thread operation order (defined by thread edges), into the new *trace-log*. Events are not created for graph helper vertices (*start*(θ) and *end*(θ)) or vertices associated with tasks that have been inlined (e.g. $v^{\text{TkExeStart}}$). An event created for a vertex, v, takes the event type and auxiliary data from the vertex's corresponding original event, $\psi(v)$, and uses the thread id and timestamp calculated for the vertex by the *trace-DAG*. Thread ids are calculated by following the thread edge paths from the special thread start vertices, *start*(θ). Absolute timestamps are calculated by propagating the relative durations on time edges through the graph. The timestamp for the event created for vertex w is calculated by:

$$\mathcal{T}(w) = \max_{(v,w,d)\in E^{\tau}} \mathcal{T}(v) + d \tag{4.14}$$

thereby satisfying all inequality time constraints. To improve estimation accuracy when trace-DAGs have been edited, the duration of thread sleep operations is adjusted by modifying duration labels on specific related time-edges before calculating the absolute timestamps, this is discussed in Section 4.2.4.5.

Formally, to derive a trace-log, define a function, $\sigma(v, \theta)$, that gives an event for the vertex v on thread θ for the new *trace-log*:

$$\begin{aligned}
\sigma(v,\theta) = \text{Event } \{ & \\
\mathit{type} &= \psi(v).\mathit{type}, \\
\mathit{aux_data} &= \psi(v).\mathit{aux_data}, \\
\mathit{timestamp} &= \mathcal{T}(v), \\
\mathit{thread_id} &= \theta \\
\}&
\end{aligned} \tag{4.15}$$

Recall that ψ is the mapping from vertex to event and $\mathcal{T}$ returns the calculated absolute timestamp for a vertex. Refer to Fig. 3.2 for the definition of the event data structure.

Let $L = \mathcal{L}(G)$ be the trace-log derived from the trace-DAG G. Then L is the concatenation of the per-thread trace-logs for all threads present in G. The threads present G are given by the special thread start and end vertices (recall the vertices *start*(θ) and *end*(θ) from Section 4.2.4.2). Write L^{θ} as the per-thread trace-log for thread θ. For simplicity I employ a set-constructor-like notation for defining the ordered sequence of events in a per-thread trace-log. Using this notation, each per-thread trace-log is given by:

$$\begin{aligned}
L^{\theta} = (\sigma(v_0,\theta), \ldots,\ \sigma(v_{n-1},\theta) \mid\ & (\mathit{start}(\theta),\ v_0) \in E^{\Theta}, \\
& (v_i,\ v_{i+1}) \in E^{\Theta}, \\
& (v_{n-1},\ \mathit{end}(\theta)) \in E^{\Theta}\ )
\end{aligned} \tag{4.16}$$

4.2.4.5 Sleep estimation

There are two types of sleeps in *Rehype*'s target execution model[10]: a worker thread waiting to be assigned a task to execute and some code waiting for the result of a task to be set

[10] A program may also include other types of sleeps, however, *Rehype* will not estimate an updated sleep duration for those sleeps.

so it can be retrieved. In a *trace-DAG* these sleeps end when a specific corresponding concurrent event occurs (*task-created* and *result-set*, respectively). This concurrent event determines the *earliest-sleep-completion-time* (ESCT). When a *trace-DAG* is edited the ESCT for sleeps may change. Thus to accurately estimate the updated *trace-log* the duration of the sleeps must be updated to match the ESCT. If the ESCT is before the start time of the sleep, the sleep will have a zero-duration and all events within the sleep (e.g. function events for `Object.wait()`) occur immediately after the sleep start event with the same absolute timestamp.

Accurately estimating the duration of sleeps requires its own step in the graph to log transformation as the normal time-edges in the trace-DAG are not able to calculate the ESCT. The normal time-edges calculate forward time durations, however, the ESCT requires calculating backward through the time-edges. This process of calculating backward through the time-edges is easiest to understand via an example.

Fig. 4.9 illustrates a trace-DAG sub-graph containing the vertices and edges of a sleep operation and the event it waits for. The sleep operation in the figure is a thread pool worker thread waiting for the next task it executes to be created. In the graph v_1 is the start of the sleep operation (`ThreadWait`), v_2 and v_3 are the FnStart and FnEnd events for a `wait()` invocation, respectively, v_4 is the end of the sleep operation (`ThreadStart`), v_5 is the `FnStart` of `FutureTask.run()`, v_6 is the `TkExeStart`, and w_2 is the `TkCtor` (w_1 is irrelevant and simply illustrates a prior event on the same thread as w_2). w_2 is the event that the sleep operation waits for. All v_x vertices occur on the same thread while w_2 is on a separate thread. The ESCT is the timestamp of v_4 such that v_6 executes immediately after w_2 (i.e. v_6's earliest possible timestamp while still satisfying all incoming time-edge constraints). Thus, in the example in Fig. 4.9, the ESCT, ε, is:

$$\varepsilon = \max(\mathcal{T}(v_1),\ \mathcal{T}(w_2) + 0 - d_5 - d_4)$$

where $\mathcal{T}(v)$ returns the absolute timestamp calculated for vertex v. To adjust the duration of the sleep, the edges within the sleep operation (i.e. $d_1,\ d_2,\ d_3$) are scaled to the ESCT based sleep duration:

$$\begin{aligned} r &= \frac{\varepsilon - \mathcal{T}(v_1)}{d_1 + d_2 + d_3} \\ d'_1 &= r d_1 \\ d'_2 &= r d_2 \\ d'_3 &= r d_3 \end{aligned}$$

To perform this sleep estimation for every sleep in a trace-DAG, sleep sub-graphs are identified. A sub-graph consists of the sleep start (v_1 from the example), sleep end (v_4), concurrent receiver (v_6), and concurrent trigger (w_2) vertices, and the paths of vertices and edges between them.

Concretely, assume a subset of all vertices V^{start} that contains all sleep start vertices and a subset V^{θ} that contains the vertices on the thread θ. Then, a sleep-path – the sequence of vertices between sleep start and concurrent receiver vertices, inclusive – on a

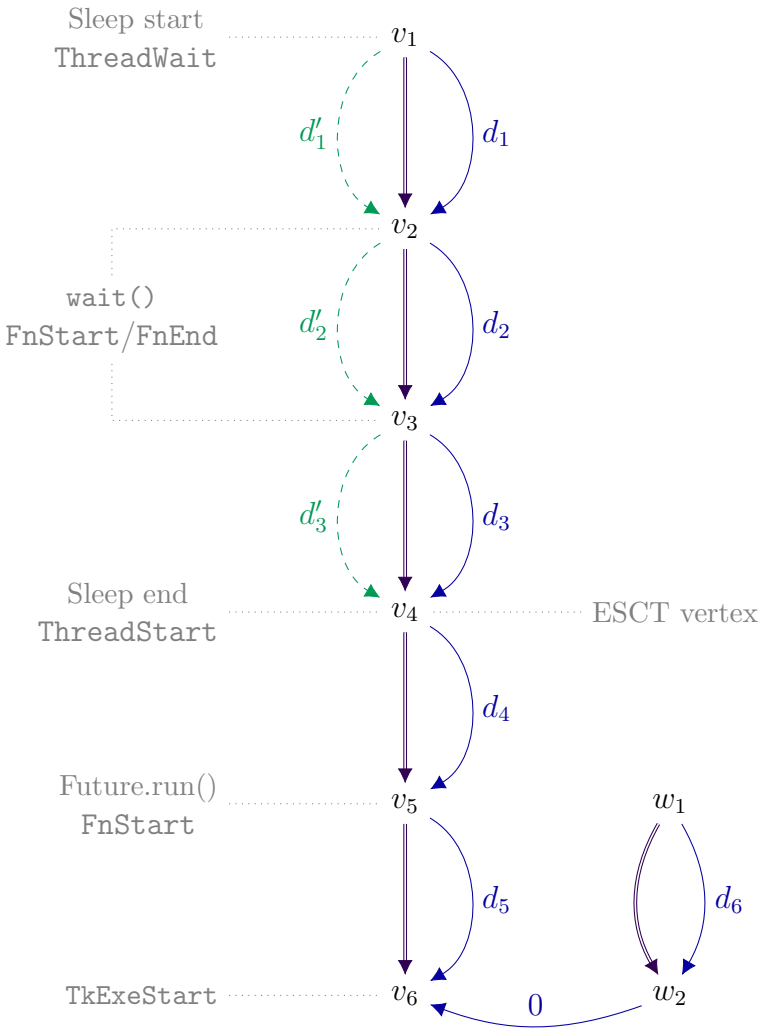


Figure 4.9: Illustration of a sub-graph containing the vertices and edges in a simple sleep operation. Blue edges are the original time-edges, green edges are the adjusted sleep time-edges (which replace their original counterparts). See the text for explanation of sleep estimation based on this sub-graph.

thread θ is given by (using the sequence notation from Eq. 4.16):

$$\begin{aligned}(v_1, \ldots, v_n \mid\ & v_1 \in V^\theta \cap V^{\text{start}}, \\ & v_i \in V^\theta, \\ & (v_i, v_{i+1}) \in E^\tau, \\ & \exists w \in (V - V^\theta)\ ((w, v_n, d) \in E^\tau)\ )\end{aligned} \tag{4.17}$$

The final vertex in the path, v_n the concurrent receiver, is the first vertex to have an incoming time-edge from a vertex on another thread (via the (w, v_n, d) edge). The ESCT, ε, is calculated as:

$$\varepsilon = \mathcal{T}(w) - \sum_{v^{\text{end}} \leq v_i < v_n}^{(v_i, v_{i+1}, d) \in E^\tau} d \tag{4.18}$$

where v^{end} is the sleep end vertex on the path and w is the concurrent trigger in the sub-graph, identified as:

$$w \mid (w, v_n, d) \in E^\tau,\ w \in (V - V^\theta) \tag{4.19}$$

Finally, scale each time edge between the sleep operation vertices:

$$|s| = \sum_{v_1 \leq v_i < v^{\text{end}}}^{(v_i, v_{i+1}, d) \in E^\tau} d \tag{4.20}$$

$$r = \frac{\varepsilon - \mathcal{T}(v_1)}{|s|} \tag{4.21}$$

$$(v_i, v_{i+1}, d) \mapsto (v_i, v_{i+1}, rd) \mid v_1 \leq v_i < v^{\text{end}} \tag{4.22}$$

here $|s|$ is the original sleep duration and r is the ratio between the original and estimated sleep durations. Recall the graph notation, $x \mapsto y \mid \textit{condition}$, replaces the edge(s) x with a new edge y, for every x that satisfies the condition (in this case, every edge connecting vertices between v_1 and v^{end}).

4.2.5 Improvement performance effect

Given the ability to estimate the effect of a change, we can automatically identify which changes have positive effects on program performance. *Rehype* implements a metric function μ that calculates a set of performance metrics for a program given a *trace-log*. A proposed change δ is beneficial if:

$$\mu(\beta_\delta \llbracket p \rrbracket) > \mu \llbracket p \rrbracket \tag{4.23}$$

Recall $\beta_\delta \llbracket p \rrbracket$ is the estimate trace-log for δ that has been reordered to approximate the trace-log that would generated if the change were implemented. The apparent benefit of δ is technically constrained to the execution of the program with the input used to generate $\llbracket p \rrbracket$. The actual benefit to the program more broadly is dependent on how representative the input is of all inputs the program processes.

The performance metrics calculated for *trace-logs* are listed in Table 4.1. There are two categories of metrics: time-series and aggregate. Time-series metrics are calculated per-nanosecond. Aggregate metrics are derived from the time-series metrics, with the exception

Metric	Description
Time-series metrics	Time-series data calculated per-nanosecond.
Active threads	Number of thread pool threads in the *active* state.
Waiting threads	Number of thread pool threads in the *waiting* state.
Unoccupied threads	Number of thread pool threads in the *unoccupied* state.
Aggregate metrics	Aggregate metrics across the whole *trace-log*. These metrics are used for determining change benefit.
Average threads utilised	Average number of threads in *active* or *waiting* state at each nanosecond. This metric determines how concurrent the program is and is an indicator to program throughput (e.g. how many requests could a server concurrently handle).
Max threads utilised	Maximum number of threads in *active* or *waiting* state at one time.
Average threads waiting	Average number of threads in *waiting* state at each nanosecond. This metric indicates how inefficient the program's concurrency is.
Max threads waiting	Maximum number of threads in *waiting* state at one time. Indicates the largest bottleneck effect in the program.
Execution duration	Time between the first and last event in the *trace-log*.

Table 4.1: Performance metrics calculated for trace-logs. Metrics are used to determine how beneficial changes are.

of the *duration* metric. Time-series metrics can be plotted to illustrate the behaviour of the program across time (e.g. bottlenecks). Aggregate metrics enable comparison of *trace-logs*. This comparison is the basis of determining the benefit of a change.

The performance effect of a change is the delta of performance metrics between the pre- and post-change *trace-logs*. The estimate effect is thus the difference between the pre-change and estimate *trace-logs*. The accuracy of estimation is the difference between the estimate *trace-log* and the post-change *trace-log*.

To sort a list of changes based on performance benefit *Rehype* calculates a single *aggregate-benefit* metric[11]. *Aggregate-benefit* is the sum of all aggregate metrics' normalised difference from the original. More formally, the *aggregate-benefit* of a change δ is:

$$a_\delta = \sum_{(e,\ b) \in (\mu(\beta_\delta [\![p]\!]),\ \mu([\![p]\!]))} \frac{b - e}{b} \tag{4.24}$$

where e is a performance metric in the estimate *trace-log* and b is its corresponding pair in the pre-change $[\![p]\!]$ *trace-log*.

4.2.6 Multiple improvements

Having described the analyser for estimating a single improvement, I now extend it to sets of improvements. Changes to a program's concurrency naturally interfere with each other. This means that the estimated effects of multiple improvements are not summative, that is, the effect of implementing them together is not equal to the sum of the effects of implementing them individually. As such, the final step of *Rehype* is to estimate the effect of each combination of improvements δ, to accurately estimate their overall effect. This can result in a combinatorial explosion of 2^n possible combinations, for n improvements.

Given the potential size of the combinatorial space, it is important to limit the space as much as possible and search it in a sensible manner. To limit the space, *Rehype* identifies sets of mutually exclusive improvements. To search the remaining space, *Rehype* uses a form of hill-climbing to more efficiently identify the most beneficial combinations. This hill-climbing optimises for a particular *composite* metric (Section 4.2.7) that minimises thread usage and execution duration, to maximise throughput.

Two improvements are mutually exclusive if they would require conflicting trace-log changes (and hence they would require conflicting source-level changes). In essence an improvement defines "do X to tasks Y", such as "do convert-to-inline to tasks A, B, and C". As an example, two mutually exclusive improvements might be: "do convert-to-combinator to tasks A and B" and "do convert-to-inline to tasks B and C". These improvements require different operations be applied to the same tasks (in this case, B), and are thus mutually exclusive.

Efficient methods for searching combinatorial space is a well established area of research and, in future work, I plan to investigate how these can be better leveraged in the improvement combinatorial space.

4.2.7 Measuring and Selecting Improvements

Rehype calculates various metrics from a *trace-log* for program p; these can be used to compare the original *trace-log* $[\![p]\!]$ with the *trace-log* $\beta_\delta [\![p]\!]$ incorporating the improvements

[11] Developers can also sort based on the individual metrics depending on their performance targets.

δ and with the *trace-log* $[\![p_\delta]\!]$ from executing the modified program p_δ. Metrics include *time-series* metrics which give a separate value for each timestamp (e.g. number of threads in each state – active, waiting, and unoccupied) and *aggregate* metrics (e.g. wall-clock execution duration and summary statistics (e.g. mean, max)) derived from time-series metrics.

The thread-usage metric values can be plotted to aid identifying bottlenecks and spikes in concurrent thread usage. The aggregate metrics can be used individually or collectively to sort and select improvements, or combinations of improvements, that best optimise for the improvements the developer seeks. In the simplest case, sets of improvements along with their associated metric values can be inserted into a spreadsheet for the developer to analyse. *Derived metrics* can usefully combine multiple metrics into a new metric. For example, squaring and averaging a set of metric values can weight greater single-metric improvements more significantly.

4.3 Evaluation

Rehype analyses trace-logs captured by *Quilt* to generate a list of suggested improvements. It provides the estimated throughput effect for each improvement.

In this section I perform a number of experiments to evaluate *Rehype*'s estimations. First, I present micro-benchmark-based experiments to illustrate the bounds of *Rehype*'s estimation, then an evaluation of the accuracy of throughput estimations, an evaluation of the consistency of estimations, and, finally, an evaluation of the throughput improvements achieved by implementing suggested improvements in an industrial application back-end server, *Acme*. The results demonstrate the effectiveness of *Rehype* for large real-world systems, the potential and validity of estimating executions based on previous executions for concurrency performance analysis, and the significant throughput improvements achievable by reducing concurrency.

I evaluate *Rehype* on a proprietary application back-end server, *Acme* (described in Section 2.6.1), which is an analogue for the Paterson software in our running scenario, as well as performing micro-benchmark based experiments to illustrate the best-case and worst-case boundaries of *Rehype*. *Acme* has extensive automatic test coverage (including unit tests and integration tests), allowing us to confirm that the improvements do not affect functional program behaviour (program outputs). In each of the presented evaluations, I execute an artificial workload that interacts with a series of endpoints within *Acme* (simulating user registration and initial user interactions). This provides a clean trace of the concurrent behaviour being analysed. There is slight variability in the performance of *Acme* across repeated executions in the evaluations. This is due to the inherent non-determinism, with regards to performance, in concurrent programs and the variability of performance of systems *Acme* depends on, such as its SQL database system.

In these evaluations I report duration and thread usage performance metrics (both from estimates and real executions) and concurrent profiles. The concurrent profile of an execution is its thread usage at each point in time. Multiple executions of a program should have close to identical concurrent profiles. Slight variations are to be expected given the non-determinism of concurrent programs. Profiles are calculated to nanosecond accuracy based on *trace-logs*. When comparing concurrent profiles we normalise them based on the execution's duration so execution speed differences do not distort the comparison when profiles are overlaid in figures.

As with other chapters, these experiments were run on a Linux benchmarking machine described in Section 2.6.

4.3.1 Micro-benchmarks

I present two micro-benchmark-based evaluations of *Rehype*'s estimation, pathological best- and worst-case scenarios, to illustrate its bounds and limitations. Both micro-benchmarks are small programs that execute a recursive function that submits a new task on each invocation. The best-case benchmark illustrates a case where the concurrency is redundant and *Rehype*'s estimation can be highly accurate. The worst-case benchmark illustrates the limitation that *Rehype* depends on a representative execution to generate accurate estimations. In the worst-case we construct a scenario where *Rehype* suggests a change, given one trace-log, that is pathologically bad when the benchmark is executed with different arguments.

The code for the best- and worst-case benchmarks is identical, the relevant recursive function is shown in Fig. 4.10. The difference between the best- and worst-case benchmarks is the value given for the `delay` parameter. In the code, we have a fixed-size thread pool and a recursive function that has two parameters `depth` and `delay`. The `depth` parameter is used to limit the recursion, it is initialised to a value such that, once all tasks are created, all threads in the thread pool will be used (i.e. it will be saturated, but not over saturated). The `delay` parameter is used to insert a sleep, of d milliseconds where d is the value of `delay`, between the recursive call and the wait on the recursive value, this is to simulate some long-running computation.

The efficiency of the concurrency within the benchmarks is determined by the `delay` parameter. If `delay` is zero (or very close to zero) each recursion will, effectively, start a task which starts a sub-task (via recursion) and immediately waits for the sub-task to complete. This is the pathological best-case for *Rehype* as the concurrency is highly inefficient and so estimating and suggesting a synchronisation of the tasks is accurate and straightforward (the synchronised version of the code is provided in Fig. 4.10b). However, if `delay` is some non-trivial value above zero, then each task will be starting a sub-task (via recursion) and then performing some computation concurrently (the sleep simulates some long-running computation) before waiting for the result of the sub-task. In this case, synchronising the tasks would be bad as the computations (the sleeps) performed in each task would become sequential instead of concurrent. Thus, the pathological worst-case for *Rehype* is performing estimation based on a trace-log of an execution with `delay` equal to zero (or very close to zero) and then having the program executed with a `delay` of some larger value. In fact, the (in)accuracy of *Rehype*'s estimation will be directly proportional to the value of `delay` – the greater the value the more significant the effect of synchronising the tasks and thus the worse the accuracy of the estimation.

Note that in each micro-benchmark evaluation, there will be at least one thread that is waiting for most of the execution. This is the main thread that invokes the first iteration of the recursive function.

All benchmarks were executed 100 times and all figures in this section illustrate averages (mean and standard deviation) over those 100 executions, with the exception of concurrent profiles which illustrate a single run. The concurrent profiles naturally illustrate a single execution and are very similar across all 100 executions.

```java
// An executor service that is used in each invocation of 'recursive'.
ExecutorService executorService = Executors.newFixedThreadPool(10);

// The core recursive function executed in the micro-benchmarks.
// The function will recurse until 'depth == 0'.
// The first invocation will receive 'depth == 9' so 10 tasks will be running concurrently
// (i.e. the thread pool will be fully saturated, but not over saturated).
Future<Integer> recursive(int depth, int delay) {
  // Submit a new task to the thread pool.
  return executorService.submit(() -> {
    // When 'depth == 0' the recursion has reached its limit.
    if (depth == 0) {
      // Sleep to simulate some long running computation.
      Thread.sleep(100);
      return ((Integer) new Random().nextInt()).hashCode();
    } else {
      // 'depth != 0' so perform recursion with 'depth - 1'.
      Future<Integer> result = recursive(depth - 1, delay);
      // Sleep for 'delay' milliseconds -- simulating some computation.
      // In the best-case scenario the delay is always zero (i.e. does not sleep).
      Thread.sleep(delay);
      // Wait for the sub-recursion to finish and return its value.
      return result.get();
    }
  });
}
```

(a) The concurrent (original) version of the recursive function.

```java
// An executor service that is used in each invocation of 'recursive'.
ExecutorService executorService = Executors.newFixedThreadPool(10);

// The core recursive function executed in the micro-benchmarks.
// The function will recurse until 'depth == 0'.
// The first invocation will receive 'depth == 9' so 10 tasks will be running concurrently
// (i.e. the thread pool will be fully saturated, but not over saturated).
Integer recursive(int depth, int delay) {
  // When 'depth == 0' the recursion has reached its limit.
  if (depth == 0) {
    // Sleep to simulate some long running computation.
    Thread.sleep(100);
    return ((Integer) new Random().nextInt()).hashCode();
  } else {
    // 'depth != 0' so perform recursion with 'depth - 1'.
    Integer result = recursive(depth - 1, delay);
    // Sleep for 'delay' milliseconds -- simulating some computation.
    // In the best-case scenario the delay is always zero (i.e. does not sleep).
    Thread.sleep(delay);
    return result;
  }
}
```

(b) The synchronised version of the recursive function.

Figure 4.10: The two versions of the recursive function used in the micro-benchmarks (written in Java).

Best Case

In the best-case benchmark, *Rehype* correctly suggests inlining the task created in the recursive function as there is no parallel computation, so it is effectively just $n-1$ threads sleeping while waiting for 1 thread to do some computation, where n is the number of threads in the thread pool.

Fig. 4.11 illustrates the concurrent profiles of the best-case benchmark, including the *base* execution which is analysed by *Rehype* to generate the *estimate* execution, and finally the *synchronised* execution which is generated by executing the benchmark with the synchronised version of the code (i.e. using the code from Fig. 4.10b instead of Fig. 4.10a). Unlike the aggregate statistics given in other figures, these concurrent profiles represent individual executions (as noted above). Fig. 4.11a illustrates the usage of *occupied* threads while Fig. 4.11b illustrates the *waiting* threads. These concurrent profiles show the recursive function starting multiple tasks (each on its own thread), performing some computation (i.e. the `Thread.sleep(100);` instruction), and then finishing each task until it returns to the base number of threads. As expected, the figures mirror each other (though the waiting threads are always 1 less than the occupied threads) since each task, except the one from the final recursion, starts a new task and immediately waits. In this benchmark, the estimate is very accurate and so the concurrent profiles for the estimate and synchronised executions overlap entirely (at the perceptible level, of course the raw data differs very slightly).

Fig. 4.12 illustrates the aggregate metrics for the best-case benchmark, with all metrics normalised against the base case value. These metrics reflect the accuracy of *Rehype*'s estimation at an aggregate level. Unsurprisingly for a "best-case benchmark" the accuracy is near perfect (with minor inaccuracy on the execution duration, which is to be expected given the inherent, minor, variability between executions).

Worst Case

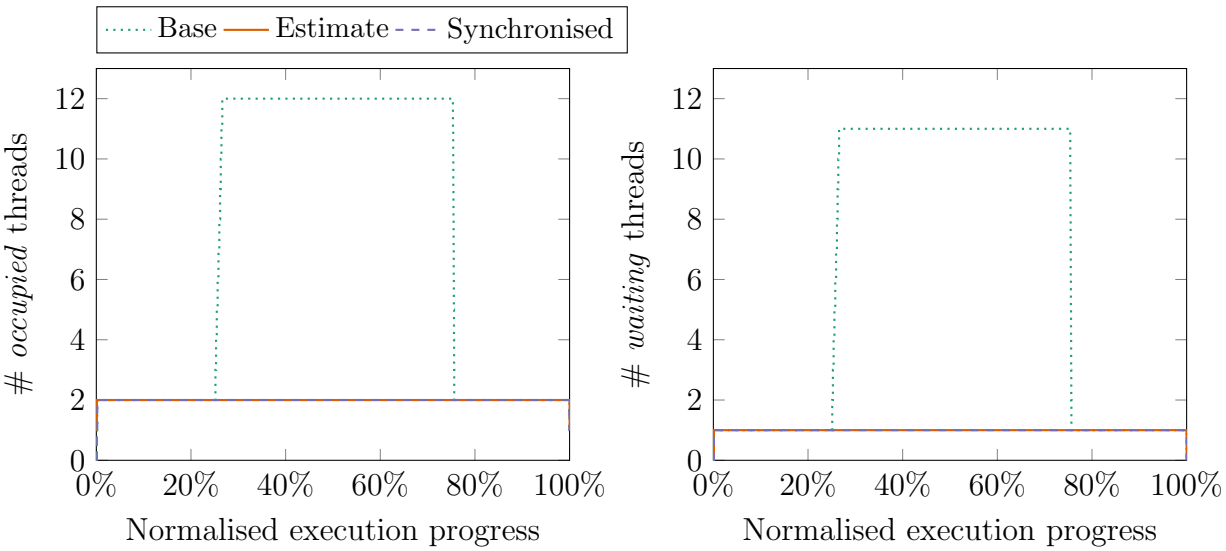
In the worst-case benchmark, *Rehype* suggests inlining the task(s) created in the recursive function as there is no parallel computation in the base execution (that estimation is performed on) – this is the same as in the best-case benchmark. However, in the worst-case benchmark, we execute the synchronised program with a different `delay` argument such that each task performs computation for some non-trivial amount of time (simulated by a `Thread.sleep()` call, see Fig. 4.10). This means that the base execution is an *unrepresentative run*, causing the estimation to be inaccurate (proportional to the `delay` argument).

To illustrate the behaviour in this micro-benchmark, this evaluation has three parts:

1. Illustration of the behaviour of the micro-benchmark given varying `delay` argument values.

2. Evaluation of the estimation when using the same, non-zero, `delay` value in both the *base* and *synchronised* executions (i.e. when estimation is based on a representative execution).

3. Evaluation of the estimation when using different `delay` values in *base* and *synchronised* executions, specifically a zero delay for the base and a 100 millisecond delay for the synchronised execution (i.e. when estimation is based on an unrepresentative execution).



(a) The concurrent profile of *occupied* threads (i.e. all threads processing a task, waiting or active).

(b) The concurrent profile of *waiting* threads. This profile mirrors the profile of occupied threads (Fig. 4.11a), minus 1 (i.e. the active thread). This shows that there is, effectively, only one thread actively working at any one time, in the base execution.

Figure 4.11: Concurrent profiles of the base, estimate, and synchronised executions (normalised for execution duration) for the best-case benchmark. This figure illustrates the estimation's accuracy at an operational level – the estimate and synchronised thread usage lines overlap entirely (at the perceptible level; they have very small variations in raw data). The space either side of the spike in thread usage is simply a sleep on the main thread of the benchmark, which is inserted to make the concurrent profile more intuitive (without the sleep the thread count lines would go up and down on the edge of the figure, making it difficult to read).

1. Micro-benchmark behaviour. The behaviour of the micro-benchmark is determined by the `delay` parameter. The greater the value the more computation each task performs (simulated by a `Thread.sleep()` call). Fig. 4.13 illustrates concurrent profiles of executions of the micro-benchmark program's concurrent version (see Fig. 4.10a), given varying `delay` argument values (no delay, 10 milliseconds delay, 50 milliseconds delay, 100 milliseconds delay, and 1000 milliseconds delay). The concurrent profiles illustrate the number of *waiting* threads, as opposed to *occupied* threads, to illustrate the relative concurrent efficiency (the less time threads spend waiting the better). The profiles show that as the value of the `delay` argument increases, the proportion of the program the threads spend waiting decreases. Fig. 4.14 shows the aggregate metrics (as seen in Fig. 4.12) for the benchmark when executed with each of the `delay` configurations.

2. Representative execution estimation. Given a non-zero (or very close to zero) `delay` argument value, *Rehype* will correctly estimate that synchronising the recursive tasks would significantly hinder performance as it would make the long running computations, that are done in parallel in the base execution, sequential. Fig. 4.15 illustrates this by comparing the execution durations the base, estimate, and synchronised executions, when

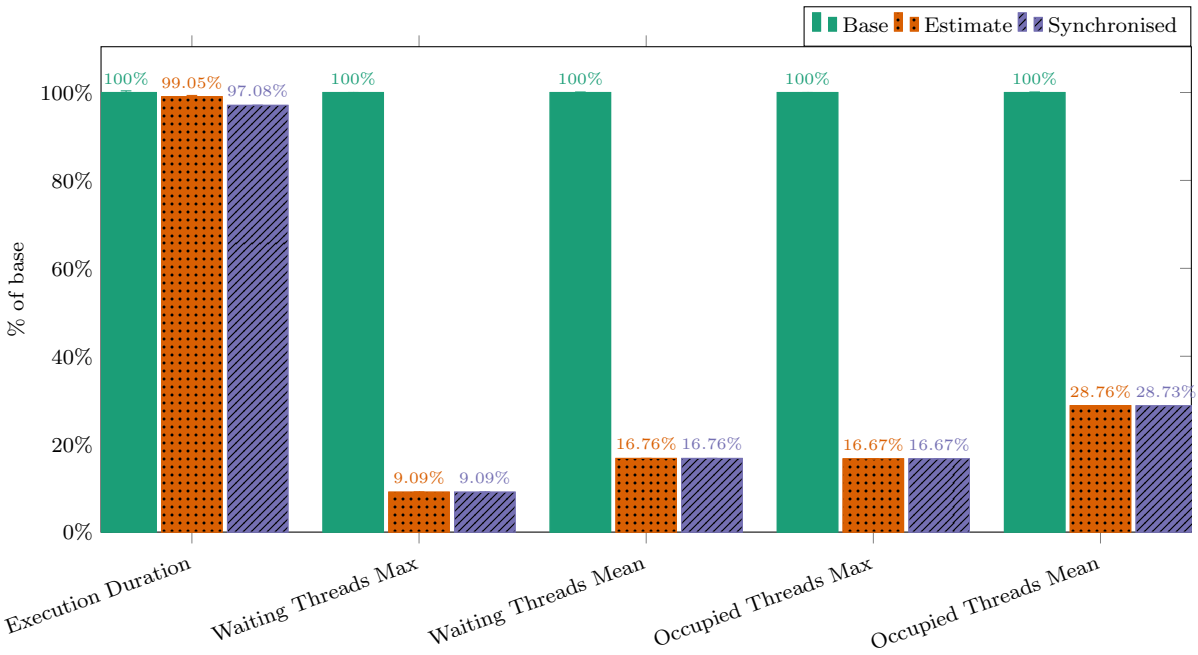


Figure 4.12: Aggregate metrics for base, estimate, and synchronised executions of the best-case benchmark, normalised to the base case. Recall that occupied threads are all threads that are currently processing a task, whether active or asleep. Error bars represent the standard deviation across runs (the standard deviations are very small, as is expected for micro-benchmarks, hence the error bars are hard to see).

the base and synchronised executions are given the same `delay` argument value (it includes the five `delay` values as above, no delay and delays for 10 milliseconds, 50 milliseconds, 100 milliseconds, and 1000 milliseconds). The key result shown in this figure is that *Rehype* will correctly estimate that performing the synchronisation of the tasks is bad, when given a trace-log of a representative execution. In this case, a representative execution is one with a `delay` argument suitably greater than zero. As shown in the figure, even given a `delay` of 10 milliseconds *Rehype* correctly estimates that the synchronisation would be detrimental to performance (i.e. it is suitably representative of executions given a `delay` greater than zero).

3. Unrepresentative execution estimation. Evaluation 2. (Fig. 4.15) demonstrated the effect of a representative execution on *Rehype*'s ability to accurately and correctly[12] estimate the effect of a change. This evaluation, illustrated in Fig. 4.16, extends upon this by showing how, within the micro-benchmark, the accuracy of estimation based on an unrepresentative execution (i.e. one using a `delay` of zero) is inversely proportional to the value of the `delay` argument. The figure compares the aggregate metrics of an unrepresentative base execution, an estimation based on it, and executions of the synchronised program (Fig. 4.10a) given various `delay` values.

[12]In this context, accuracy can be thought of as a continuous value, while correctness is a boolean – either an estimate is correct or it is incorrect. If *Rehype* suggests a change would be beneficial or detrimental and it is (based on implementing the change and executing the program again), then it can be considered "correct". Whereas, if *Rehype* suggests a change is beneficial or detrimental and it is not, it can be considered "incorrect". However, the accuracy of an estimation can vary substantially whilst the

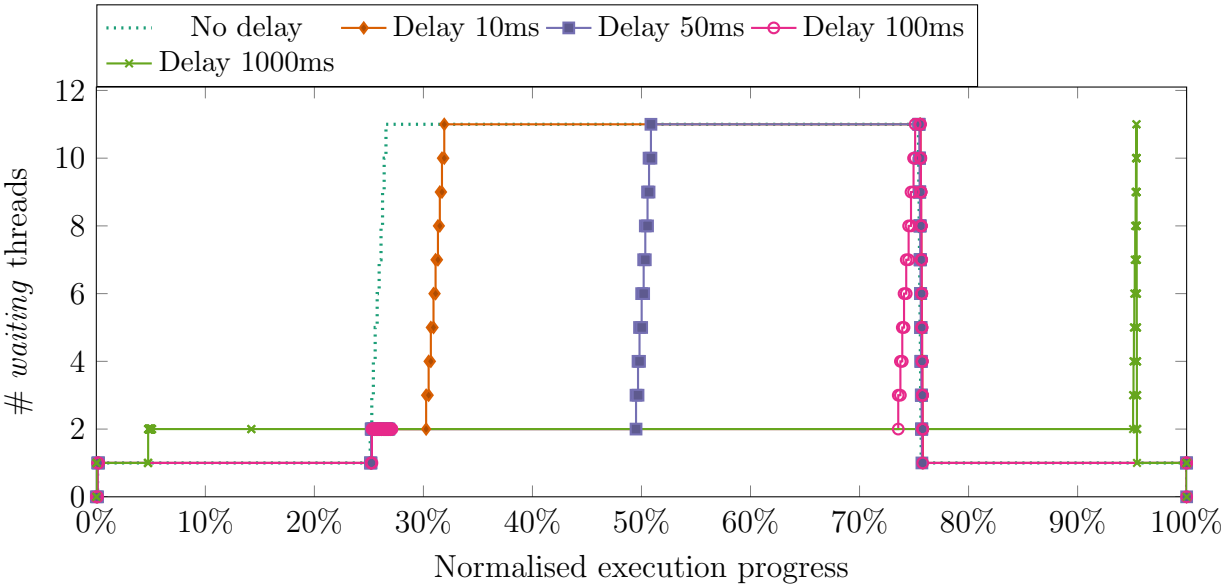


Figure 4.13: Concurrent profiles of *waiting* threads in the micro-benchmark's concurrent version (see Fig. 4.10a) when executed with five different `delay` argument values. In the "No delay" execution, each task recursively creates a sub-task and then immediately waits for it, the deepest task then sleeps for 100 milliseconds and the remainder wait for it to be finished (hence the table-like pattern). In the "Delay x ms" executions, each task creates its sub-task and then sleeps for x milliseconds, before waiting for its sub-task. In the "Delay 10ms" and "Delay 50ms" executions, the recursive tasks each sleep for the set number of milliseconds (based on `delay`) and then wait for approximately $100 - d$ milliseconds for the deepest task (which always sleeps for 100 milliseconds, Fig. 4.10a) to finish, where d is `delay`. In the "Delay 100ms" execution, the recursive tasks only briefly wait near the end of the execution as all of the tasks should finish at approximately the same time (the extra waiting time is due to the delay between submitting a task to the thread pool and it being executed). Finally, in the "Delay 1000ms" execution, the waiting is determined by the same task submission to execution delay, but is very short relative to the overall execution time. Recall that concurrent profiles represent thread usage over *normalised execution progress*, hence the difference in shape at the start of the execution (all of the executions have a short setup period, this period simply appears shorter for "Delay 1000ms" as the execution's overall duration is longer).

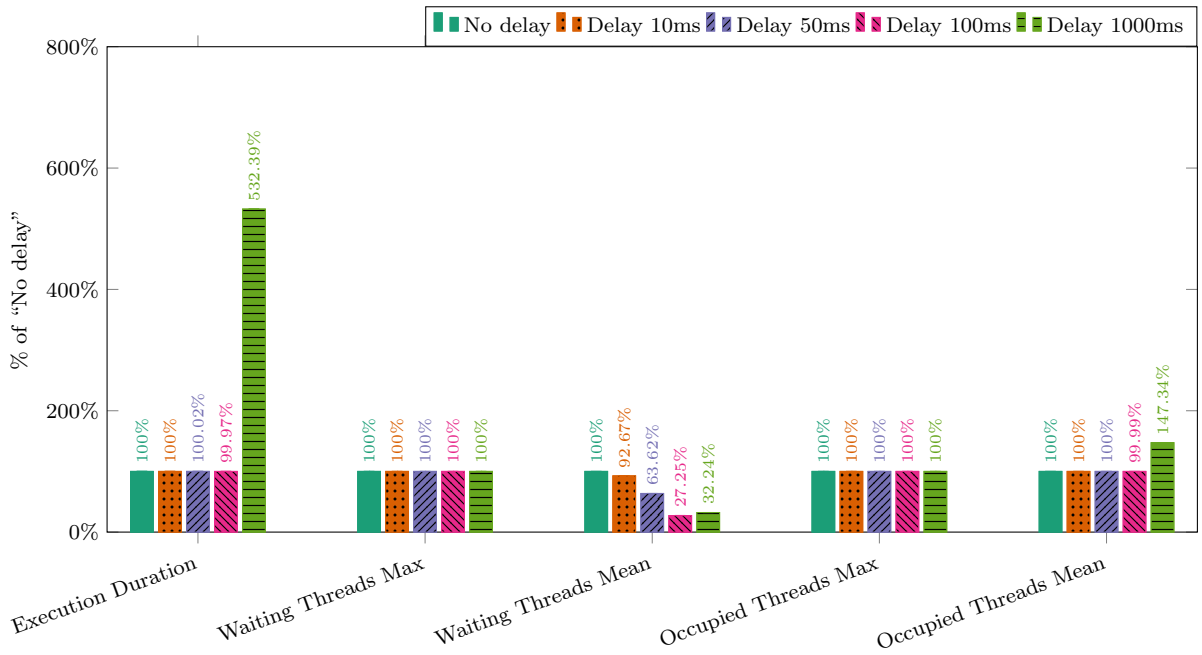


Figure 4.14: Aggregate metrics for executions with varying `delay` argument values. The metrics are normalised to the "No delay" execution. There are two interesting aspects to this figure: first, the mean number of waiting threads decreases as the delay increases, as threads are waiting for less time; and second, the execution duration does not significantly change until the delay exceeds 100 milliseconds as that is the time the final recursive task waits for (it is a constant, see Fig. 4.10a). Error bars represent the standard deviation across runs (the standard deviations are very small, as is expected for micro-benchmarks, hence the error bars are hard to see).

Summary

These micro-benchmark evaluations demonstrate the bounds of *Rehype*'s estimation, the necessity of trace-logs from representative executions, and provided an introduction into the key metrics and visualisations used in the remainder of this chapter. As Fig. 4.15 and Fig. 4.16 illustrated, performing estimation on an unrepresentative execution leads to inaccurate, and potentially incorrect, estimations. However, given a representative execution, *Rehype* is able to correctly and accurately estimate the effects of potential changes. In the next experiments, we will look at *Rehype*'s ability to estimate change effects in a non-trivial program.

4.3.2 Estimation Accuracy

Estimation accuracy can be assessed based on an execution's concurrent profile and an execution's high-level concurrency performance metrics. Concurrent profiles can provide insight into the effects of improvements throughout an execution and validate the accuracy of the estimator. While high-level performance metrics provide clear indicators of change value that can be compared between changes and also demonstrate the benefit of a change for the cost-benefit of a developer implementing a change.

correctness remains unchanged.

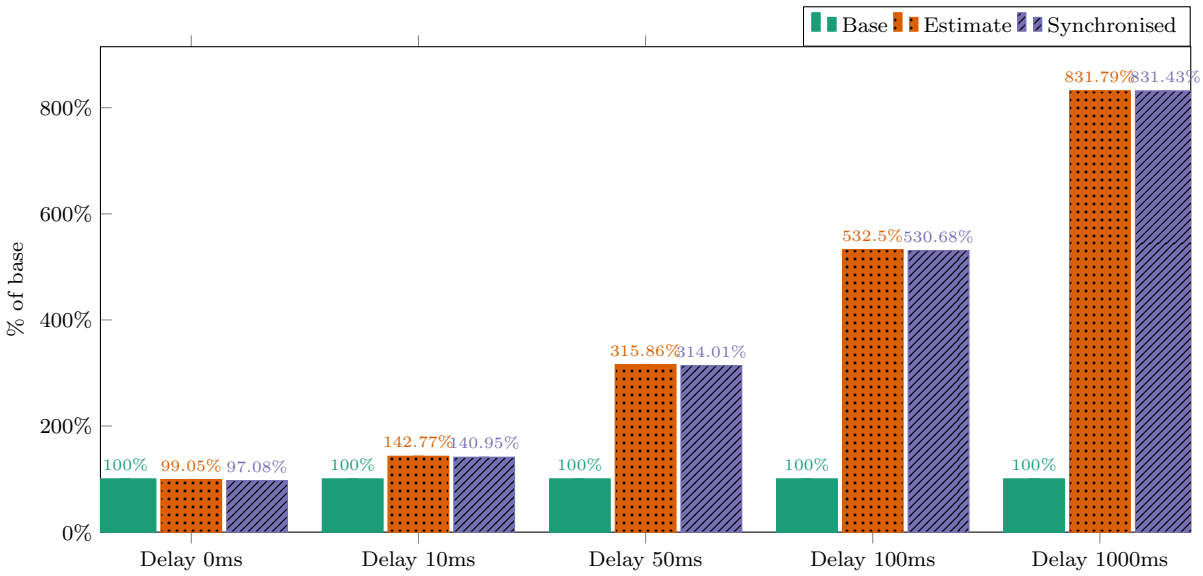


Figure 4.15: Execution duration for base, estimate, and synchronised executions of the worst-case micro-benchmark, given five different `delay` argument values. The values are normalised to the base execution duration, for each `delay` value configuration. This figure illustrates *Rehype*'s ability to estimate slowdowns due to synchronisation, when given a representative execution. In this experiment, we use the same `delay` value for the base and the synchronised executions, so it is exactly representative. However, the key point is that *Rehype* would suggest *not* making the synchronisation change in all cases except the "Delay 0ms" case. So, for example, "Delay 10ms" is suitably representative so as to make *Rehype* suggest not making the change, even though the slowdown is not as significant as the "Delay 1000ms" execution. Error bars represent the standard deviation across runs (the standard deviations are very small, as is expected for micro-benchmarks, hence the error bars are hard to see).

Given a set of changes, comparing the concurrent profiles of an estimated *trace-log* of the changes and a real *trace-log* from an execution of the program after implementing the changes demonstrates the accuracy of estimation across the execution of a program. Concurrent profiles also provide insight into the areas within executions that changes affect. For example, a change might affect the start up process and not the main body of operations in the program.

The changes evaluated in this section are *convert-to-inline* and *convert-to-combinator* changes made to a cryptography module of *Acme* (used to encrypt and decrypt data, and hash passwords). The artificial *Acme* workload used for evaluation contains multiple requests to varying endpoints.

Fig. 4.17 illustrates three concurrent profiles of *Acme*, a base execution, an estimation of changes from the base *trace-log*, and an execution of *Acme* with the changes implemented. The key aspects of the comparison to note are:

1. The estimated and *actual-change* profiles are very similar, including specific variations in profile from the base profile.

2. They have a slight temporal displacement at the start of their respective executions. Temporal displacements such as this are affected by the approximate nature of estimations and the non-determinism of concurrent programs' performance.

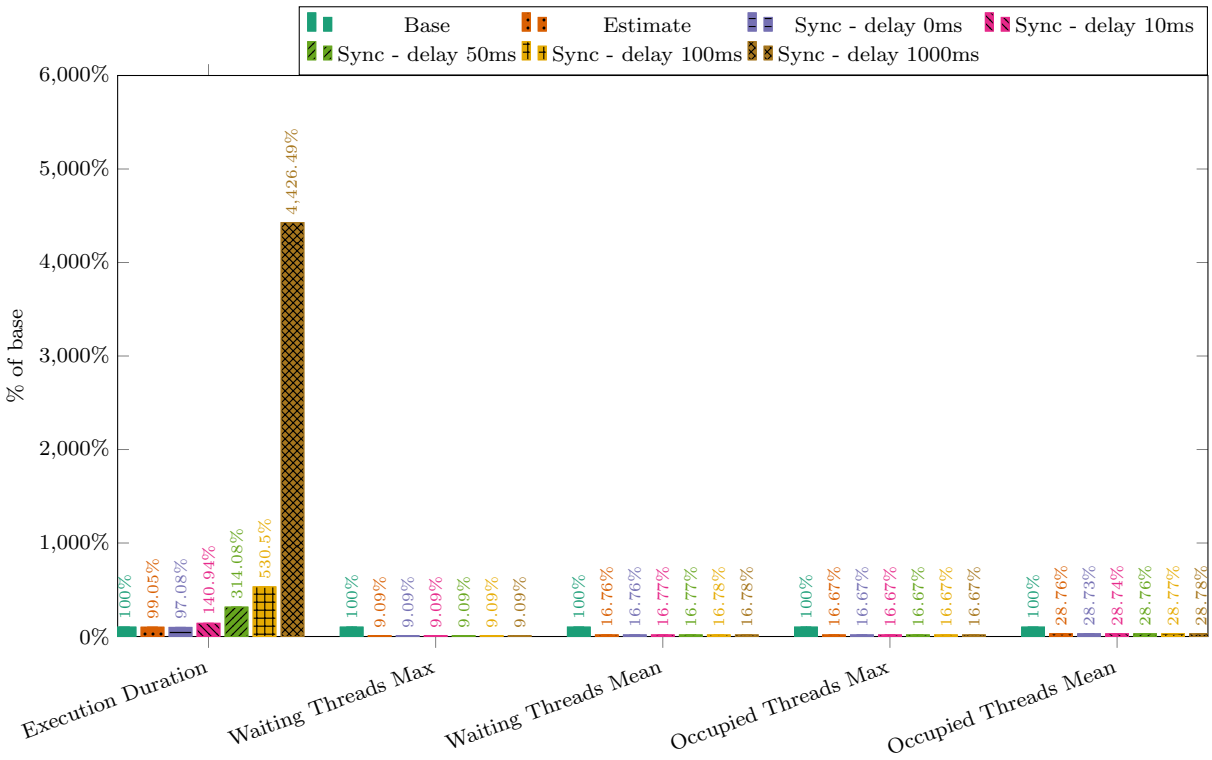


Figure 4.16: Aggregate metrics comparing the base execution with zero delay, an estimate based on that execution, and executions of the synchronised program (Fig. 4.10a) given various `delay` argument values. This figure shows that as the value of the `delay` argument increases, the accuracy of the estimation decreases proportionally (as the distance between the estimate values and the actual values increases). In this micro-benchmark it is most evident in the "Execution Duration" metric (the thread usage metrics remain essentially constant given the simplicity of the program). Error bars represent the standard deviation across runs (the standard deviations are very small, as is expected for micro-benchmarks, hence the error bars are hard to see).

3. The estimated profile matches the actual-change profile with regards to which stage of an execution it differs from the base profile (and thus accurately identifying where changes affect the profile).

In Fig. 4.17 the top (dashed) magnified circle highlights the estimated profile matching the actual-change profile at the effect of a change (as the base profile differs, it is clear the difference is due to a change). While the bottom (red) dotted magnified circle highlights an area unaffected by the changes, where all three profiles have a similar shape. The bottom (red) dotted magnified circle also demonstrates the second point, the estimated and actual-change profiles have a slight time offset. In these profiles, it appears the actual-change execution is faster at the beginning and slows down near the end. The profiles realign near the end of the executions, around 90% complete, as outlined by the grey dotted box. Finally, the estimated profile matches the actual-change profile with regards to where the changes' effects are reflected. Specifically, the changes are primarily reflected near the start of the execution, such as in the top (green) dashed magnified circle. This is expected as the changes are to a cryptographic module which is used primarily at the start of the request processing when the various sensitive data (e.g. passwords) are

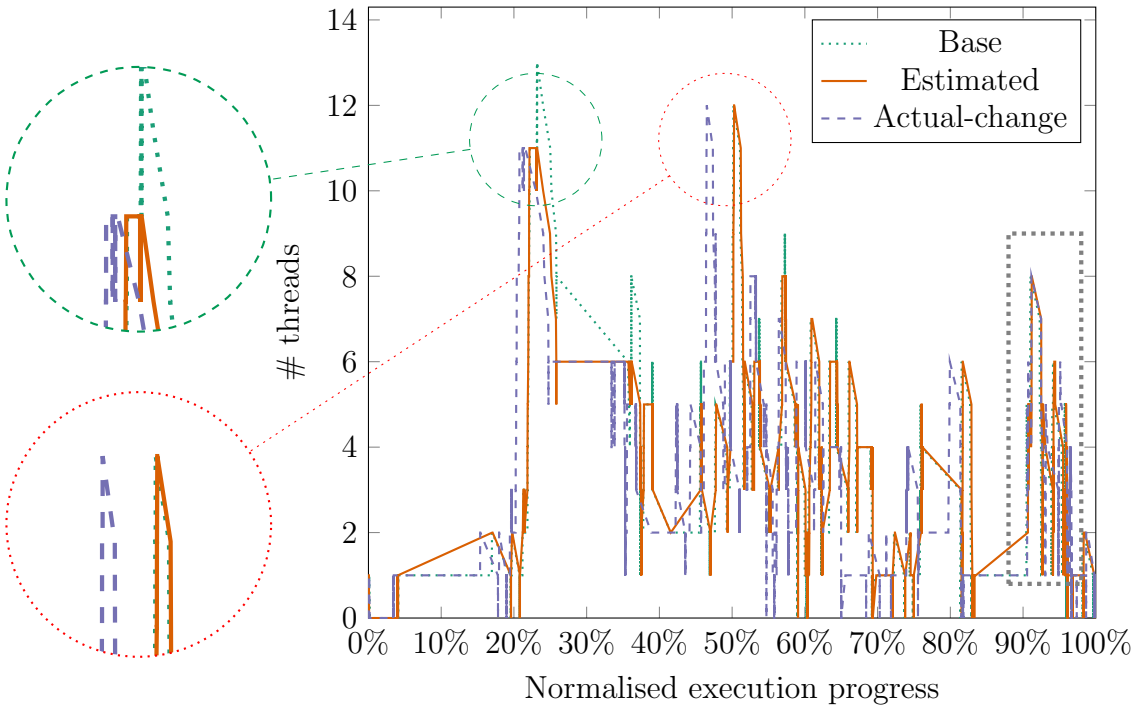


Figure 4.17: Thread usage profile over the duration of base, estimated, and actual-change executions (normalised for execution duration). This figure illustrates the estimation's accuracy at an operational level (the estimated and actual-change thread usage lines mirror each other's shape, though slightly temporally displaced).

hashed during the user registration request.

Fig. 4.18 contains aggregate thread usage metrics for each of the executions, base, estimated, and actual-change. These metrics demonstrate that the estimated execution has very similar aggregate concurrency properties to the actual-change execution. The max number of active and waiting threads used in both executions are the same. The average thread usage is similar between the executions. Indeed, the estimated execution slightly *underestimates* the efficiency improvement achieved by the changes. It overestimates the average number of waiting threads and underestimates the average number of active threads. In total it overestimates the number of threads used by the program post-changes.

The efficiency improvement demonstrated in Fig. 4.18 is achieved with few, relatively minor, changes. These changes are useful for evaluating the accuracy of the estimator with regards to aggregate metrics. That being said, the improvements are modest in comparison to those achieved by implementing a larger set of changes, as described in the next section.

4.3.3 Estimation consistency

Now we turn to evaluating the consistency of *Rehype*'s estimation on a large piece of software, *Acme*. In this evaluation, I present *Rehype*'s estimation of changes for *Acme* across multiple, varied, workloads. This relates to the previous accuracy experiment by demonstrating the estimations evaluated in that experiment are, indeed, representative of estimations on *Acme* given other workloads.

This evaluation compares *Rehype*'s change estimates across 50 workloads. Each workload performs a random series of 50 requests against an *Acme* server. Each workload

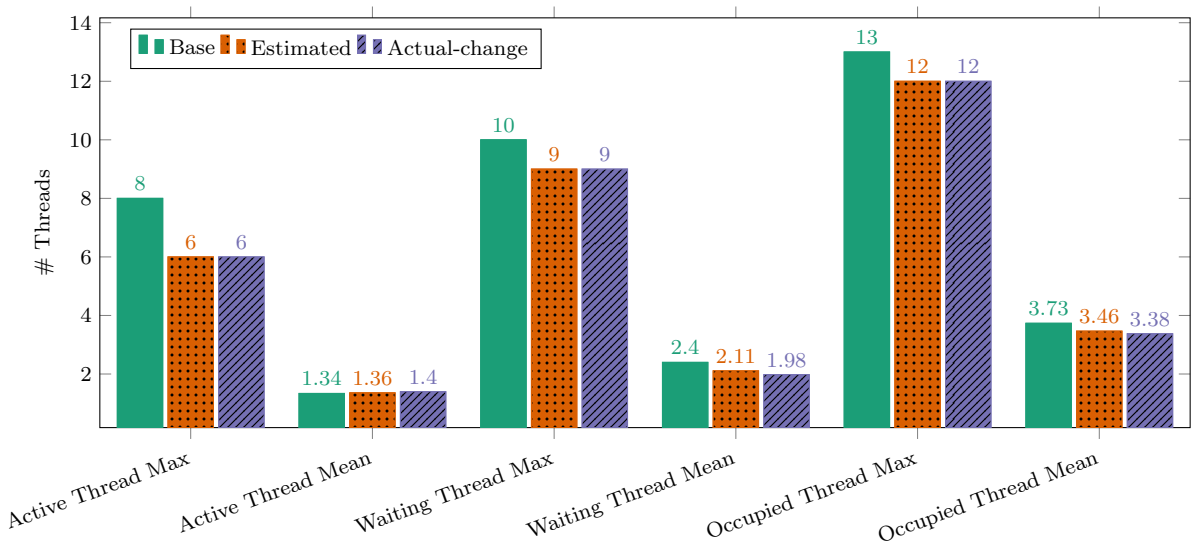


Figure 4.18: Thread usage metric comparison between a base run, an estimation of suggested changes, and a run with the changes implemented. This figure demonstrates the accuracy of the estimation with regards to aggregate metrics. Greater similarity between the dotted orange and purple striped bars is better. Execution duration is not included as it does not change significantly and is based on a different unit (time not thread count). Moreover, thread usage is more informative from an accuracy perspective, as it relates to the accurate estimation of thread usage throughout the execution of a program.

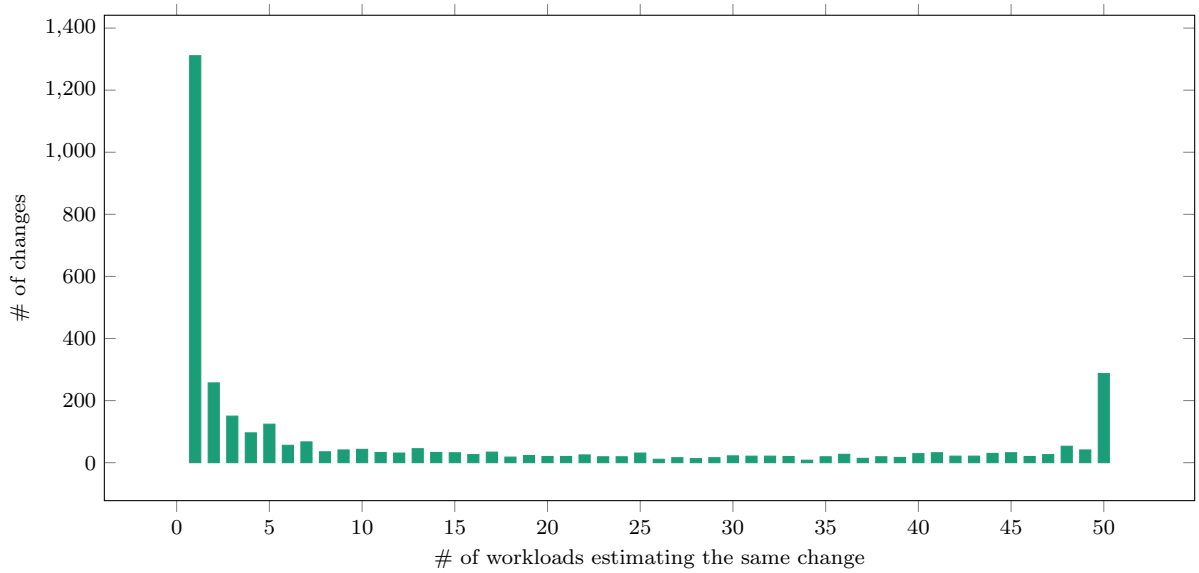
also inserts random delays (between 0 and 1,000 milliseconds) between requests. By performing a random series of requests in each workload, we can observe a variety of paths through *Acme*'s code.

Fig. 4.19 illustrates *Rehype*'s estimation consistency across the workloads in two charts. The first, Fig. 4.19a, is a histogram that shows how many changes were identified across workloads. The two points of interest are that there were 287 changes commonly identified in all 50 workloads, and that there is a long tail of changes identified in subsets of workloads, including $1,311$ changes only identified by a single workload (i.e. not commonly identified by two separate workloads). This long tail of changes is to be expected as a change can be defined by a highly specific[13] task group (see Section 4.2.2). The second chart, Fig. 4.19b, is a heatmap that illustrates the estimate improvement[14] (as the *aggregate benefit*, see Eq. 4.24) for each change within each workload. This shows the consistency of improvement estimates across workloads (as vertical stripes within the heatmap), and the tendency of some workloads to estimate a bigger improvement than others (as horizontal stripes within the heatmap). The heatmap only includes changes that were identified by 40 or more workloads to reduce noise in the figure[15]. While each change is estimated the improve performance slightly (up to $\sim 10\%$), the significant improvements are achieved by combining changes (Section 4.3.4).

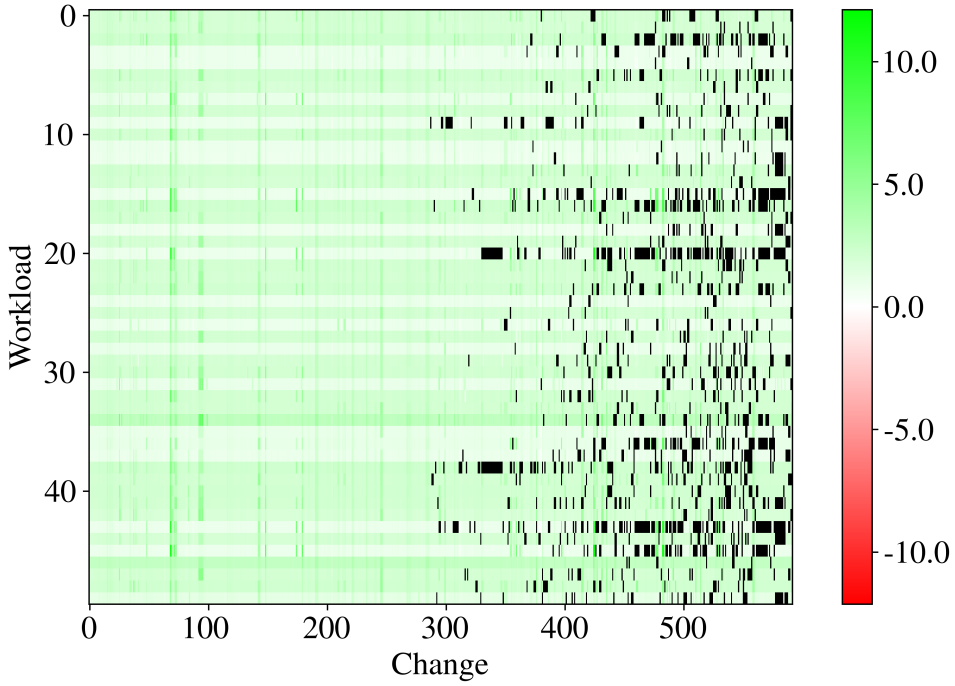
[13] At some level of specificity, it likely becomes nonsensical to implement (e.g. a change defined by a 50 function-long call stack). Whilst I do not address this question in this thesis, it is interesting to consider how we might identify such changes and exclude them from results from tools similar to *Rehype*.

[14] Whilst the heatmap shows all changes as being estimated to be positive, this is primarily an artefact of *Acme*. As shown in Section 4.3.1, *Rehype* does estimate negative results.

[15] As the number of workloads identifying a change decreases, the heatmap becomes visually dominated by black cells (which represent a workload that does not identify the change).



(a) Histogram of the number of changes that were identified by a given number of workloads. For example, 287 changes were identified by all 50 workloads, while 1,311 changes were only identified by a single workload (not the same workload). The large number of changes identified by few workloads (e.g. 30 or less) reflects the level of specificity at which *Rehype* may define a change (i.e. how detailed the dynamic context is for the change).



(b) A heatmap of improvement estimation made by each workload for each change. The improvement value is the *aggregate benefit* calculated for each change by each workload (see Eq. 4.24). This heatmap only includes changes that were identified by 40 or more workloads (the fewer workloads that identify a change, the less interesting the change is to us for this figure). A black cell indicates that the workload did not identify the change.

Figure 4.19: Experimental results from running 50 workloads and estimating changes based on the workloads.

Limitations It is important to note that these results only include changes that *Rehype* could estimate, it does not include all possible changes to *Acme*. There are many reasons why *Rehype* may not be able to generate an estimate for a change, including the code not being executed, the graph of tasks making it impossible to estimate the effect of inlining a task, or simply a bug in my implementation *Rehype*. As such, these results do not, necessarily, indicate that *Acme*'s concurrency usage is quite as inefficient as the figures may suggest, but rather that *Rehype* is able to identify a substantial number of potential improvements. Moreover, many changes will be subsets of other changes. For example, given a concurrent task, it may be possible to directly inline it, convert it to a combinator, or hoist numerous branches from its execution body, all of which would constitute individual change estimates (and, therefore, a unique change in Fig. 4.19).

4.3.4 Suggested Improvements

I evaluate the performance improvement achievable by implementing a combination of suggested changes for a program. I trace and analyse an artificial workload on *Acme* that is representative of general concurrency usage in *Acme*.

Rehype identifies 190 potential changes. Of these, 59 changes are "purely positive" (the estimate effect improves all metrics considered). Estimating the effect of 10,000 random source-change combinations, we identify a combination of 19 improvements that provide the greatest estimate *aggregate benefit* (see Eq. 4.24).

Implementing the 19 improvements we more than double the average throughput of the endpoints. That is, twice as many requests to the endpoints included in the artificial workload could be processed within the same time period. The relative changes in thread usage and execution duration are given in Fig. 4.20.

Importantly, the *effective throughput* (the throughput observed in the production environment) increase may be even greater given the substantial decrease in maximum thread usage. Maximum thread usage is the total number of threads used concurrently by the request at any time point. This is especially relevant to effective throughput as thread pools have a limited number of threads[16]. If, or when, the thread pool limit is reached, request execution can significantly slow as new tasks must be executed synchronously until another thread is available. Thread pool limits can be reached quickly if multiple requests spike in thread usage at the same time. For example, *Acme* uses a single thread pool with 20 threads to service all requests. The artificial workload evaluated caused *Acme* to have a maximum thread usage of 13 threads before the suggested-changes were implemented, post-changes it uses a maximum of 5 threads. Thus, post-changes, *Acme*'s thread pool can service 4 parallel requests hitting their maximum thread usage at the same time without exceeding its thread limit, while pre-change the thread limit would be exceeded by 2 parallel requests.

I report execution duration[17] and three sets of average and maximum thread usage metrics to demonstrate the throughput benefits and concurrency efficiency improvements. The relevance of each metric reported in Fig. 4.20 is as follows:

[16]Excluding infinitely expandable thread pools which are impractical in many production systems as they can consume too much of the system's resources, potentially crashing the program.

[17]Execution duration is reported as *Rehype* treats the trace-log as a single unit. Furthermore, the execution duration provides a single measure of the effect of changes over all aspects of *Acme* that were executed – thus providing an indicator of the overall effect of a change (from a speed perspective).

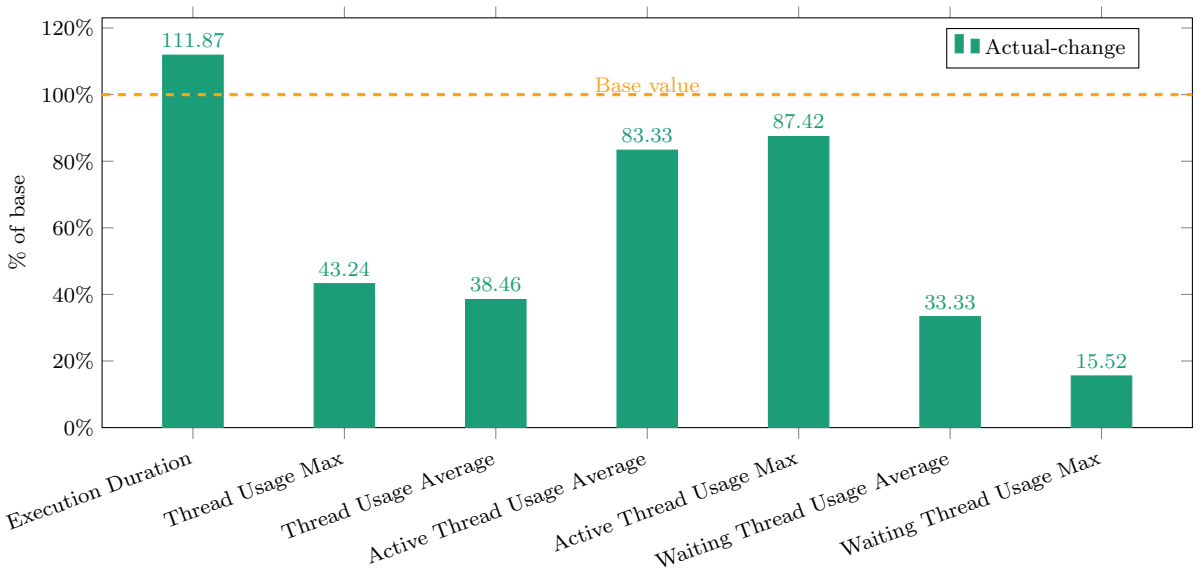


Figure 4.20: Performance improvement achieved by implementing improvements suggested by *Rehype* on *Acme* (lower is better). Values are relative to the base run of the program. While the duration increases slightly, the thread usage decreases dramatically, improving throughput, and the number of waiting threads decreases even more, indicating significantly improved concurrency usage efficiency. The result is that the program's concurrency is significantly more efficient and the program has substantially increased throughput.

Execution duration: execution duration should ideally not be substantially increased. However, it is unlikely that the changes *Rehype* suggests will significantly decrease execution duration, but rather leave it mostly unchanged while improving the other metrics. Furthermore, execution duration and, to a lesser extent, the thread usage metrics vary between executions given the non-determinism of concurrent programs with regards to performance.

Average and maximum threads: average thread usage (for each thread metric reported) is the general resource consumption of the program and thus the most direct indicator of effective throughput. Maximum thread usage, however, represents the worst case affect on throughput as it is the largest affect on thread pool limits (as discussed above, if the thread pool limit is exceeded it can have disproportionately negative affects on throughput).

Total thread usage: the total thread usage metrics provide the overall affect on throughput from the concurrency usage. Combined with execution duration, these metrics indicate the throughput achievable.

Active and waiting thread usage: these metrics indicate the efficiency of the concurrency usage within the program, as opposed to the general affect on throughput. As the number of waiting threads increase, the efficiency of concurrency decreases. Conversely, as the number of active threads increase, the efficiency of concurrency increases. In a "perfectly efficient" program there would be no waiting threads and thus the active thread metrics would be equal to the total thread metrics.

4.4 Related work

To the best of my knowledge there is no existing work that detects concurrency focused changes and estimates their performance effects. I discuss related work in two primary categories:

Performance prediction – methods for predicting the performance of programs given some change (source change or environmental change). This area relates to the estimation of source change performance effect.

Concurrency analysis – the various tasks and approaches within the concurrency analysis space, including both static and dynamic analysis methods. This relates to the complexities of analysing concurrency.

4.4.1 Performance prediction

Predicting the performance of a program given some change, be that a source change or modification of execution environment (e.g. data centre configuration), enables the developer to make informed decisions regarding the change before committing resources to the change. *Rehype* differs from existing methods as it identifies specific source changes and estimates their affect on program performance, whereas existing methods either predict the *potential* for performance improvement in a program [116] or predict performance a program given a new environment or configuration [110, 105, 93]. Furthermore, *Rehype* uses a single execution of the target program to estimate performance of many different changes, where existing methods either re-execute the program with slight variations [116] or attempt to create a model of the program behaviour for performance [93].

Performance prediction methods exist on a spectrum of specificity. The spectrum ranges from executing the real program with slight modifications, to machine learning models that regress performance based on environmental parameters. The specificity of prediction also correlates with the type of output and required level of developer knowledge. Output types range from low-level specific source changes to high-level system configurations. Required developer knowledge ranges from in-depth knowledge about the performance characteristics of the program and specific functions, to a high-level, threshold, understanding of the program that enables configuring the program. At the low-level specific end of the spectrum there are low-level performance engineering tools that enable developers to predict the effect of some change or optimisation on broader program performance, without having to implement the actual change(s). At the other end of the spectrum are machine learning methods for estimating the resource requirements of cloud software. I discuss three methods that illustrate the spectrum: first, simulating a function-level optimisation; second, simulating the performance of a whole program using a skeleton; and third, modelling a parallel program's performance based on its communication paradigm.

Dynamic performance stubs, introduced by Trapp [116], inserts a stub in place of a component of a program (e.g. a function or set of functions). To enable the program to operate with a stub, the stub returns outputs mapped from inputs (based on a recording of the functional outputs of the original component). The stub is parameterised with performance characteristics such as CPU [120, 117, 119], memory [118], and cache usage [116]. The parameters are tuned to simulate the function's performance given different levels of optimisation a developer may implement. The program is run for each set

of parameters to find the optimisation bounds of a component. The optimisation bound is the point at which further optimisations of some component do not improve the broader application's performance. By identifying the optimisation bounds of a component Trapp et al. suggest developers can better evaluate the cost-benefit of some optimisation and also identify "hidden" bottlenecks (i.e. bottlenecks that become apparent once an earlier bottleneck is optimised).

Such simulation approaches based on performance parameters (e.g. memory usage) provide an idealised view of the optimisation and do not inform how such an optimisation might be achieved. Tools based on this approach can be useful for performance engineers with a deep understanding of their target system, a relevant but niché domain. However, they are not as applicable to general performance optimisations implemented by general developers.

Where simulation approaches re-execute programs, constraining their ability to test many parameter sets, and test performance parameters, *Rehype* estimates concrete changes without re-executing the program. However, the two approaches have different target uses and audiences. Simulation-based tools are useful for manual investigation of specific optimisation opportunities. *Rehype*'s refactor-based estimation approach is useful for automatic detection and testing of concrete changes across a whole application.

Sodhi et al. [105] describe *performance skeletons*, "synthetically generated short running program(s) whose execution time always reflects the performance of the application it represents" [105]. Performance skeletons are used to predict the performance a program in a new environment (e.g. system configuration, data centre cluster, etc). Skeletons are automatically generated based on execution traces of the target program. Skeletons capture the performance behaviour of a program by utilising system resources (CPU, memory, disk IO, and network IO) in relative proportion to the target program. Sodhi et al. [105] report performance prediction accuracy of up to 95% using skeletons that execute for 5 to 10 seconds.

At a higher level, some methods model the communication mechanism of a program, such as MPI [20], to model the performance and behaviour of a program given a new environment or configuration. From message-based systems comprised of multiple decoupled components [93] to tightly coupled parallel scientific applications on high-performance-computing clusters (HPCs) that use defined communication frameworks such as MPI for parallelisation [125]. Tarvo & Reiss [112] go further, instrumenting lock and queue operations to further model the parallel behaviour of multi-threaded programs. These approaches have the advantage of clearly defined operations and structural frameworks to automatically construct models from. However, these approaches are also constrained in their level of insight into the program. They are useful for performance prediction given a new configuration where the communication primitives are the most affected (such as cluster allocation). But predicting the affect of a change in the program is harder.

4.4.2 Concurrency analysis

Most program analysis research focusing on concurrency addresses concurrency-based bugs such as race conditions [58, 9, 16], deadlocks [9], and the more general thread-safety-violations [57, 17]. Concurrency bugs, by their nature, are difficult to manually detect, reproduce, and resolve. Within this space approaches range from static analysis and symbolic interpretation, dynamic analysis and targeted program-modification (e.g. inserting

sleep instructions), through to automatic generation of tests that are executed many times attempting to trigger a bug.

Though bug detection is not directly related to my present work on performance, by reviewing this area I aim to illustrate a number of the key challenges in concurrency analysis more generally and how existing work addresses these challenges. While static analysis is quite distinct from dynamic performance analysis, such research provides insight into the broader nature of concurrency. Dynamic analysis techniques, on the other hand, are more closely related and can be useful comparisons with regards to data considered and analysis techniques.

Many approaches to concurrent program analysis either do not scale well, miss significant instances, report false positives, or a combination thereof. These issues arise due to the central challenge of concurrency analysis, non-determinism of behaviour. Static analysis techniques for identifying relations such as points-to or happens-before are constrained by the complexity of detecting these relationships in large programs. Static analysis approaches based on thread interleaving must address the challenge of a massive combinatorial space of potential interleavings. In contrast, dynamic analysis based approaches require multiple executions of the program to trigger potential bugs. Static analysis tools can generally detect bugs faster, though they report false positives, while dynamic analysis tools are slower but generally do not report false positives [49].

Li et al. [58] describe SWORD, a static analysis method for detecting data races across whole programs. The key contribution is scaling static analysis to large programs. They combine points-to and happens-before analysis to detect races while achieving speed by focusing on typical race patterns and excluding utilities, such as incremental analysis, that can impair scalability. Bläser [9] describes a static analysis method that takes inspiration from dynamic analysis by performing "randomised bounded concrete concurrent interpretation" to identify potential data races and deadlocks efficiently. They optimise for efficiency as it enables the analysis to be used during development. Bounded the interpretation is key given the combinatorial explosion of potential concurrent execution pathways.

Li et al. [57] present a dynamic analysis method, TSVD, for detecting thread-safety violations. Specifically, they trace thread-unsafe methods, rather than synchronisation operations, to identify potential violations. To verify violations they inject delays to encourage the violations to be triggered. Chen et al. [16] use dynamic analysis to identify potential data races in device drivers by tracing driver and variable accesses. Their detection is based on inconsistent lock protection, whereby the same resource is accessed with varying sets of locks throughout the program. Similarly Cai et al. [13] use dynamic analysis to identify concurrent events that could be reversed in a different execution and thus cause concurrency errors. Specifically they detect use-after-free, null-pointer-dereference, and double-free errors. Deng et al. [23] describe a coverage metric to guide detection of concurrent bugs using existing detectors. They attempt to make detection more efficient by identifying overlapping, in terms of coverage, executions caused by inputs.

One approach to combining static and dynamic analysis is to identify potential errors with static analysis and then verify these errors with dynamic analysis. For example, Zhang et al. [127] use static analysis to identify potential failure and propagation locations, and then use dynamic analysis to identify the triggers for these failures and verify the errors. This reverses the standard bug process of initial bug trigger, propagation of the error through the system, and finally a system failure caused by the bug.

Test generation A general approach to concurrency bug detection is automatic test generation, whereby the method uses some strategy to generate automated tests to (hopefully) trigger the intended concurrency bug(s). Generated tests are then run repeatedly until they discover a concurrency bug or hit some threshold (e.g. execution time). Test generation strategies include: random, sequential-test-based, and coverage-based. The random strategy selects methods at random to call in concurrent threads [77, 91]. Sequential-test-based strategies combine multiple existing sequential tests into a single concurrent test [101, 100, 102]. Coverage-based strategies generate tests to invoke as much of the target concurrent code as reasonably possible [109, 113]. Some methods combine these strategies; Choudhary et al. [17] use a novel test coverage metric to guide a random test generation strategy. The key limitation of test generation for concurrency bug detection, is that it requires an unbounded number of executions of the program. For small programs, or sub-components of a program, this is feasible, however, for larger systems multiple re-executions may be prohibitively costly. More broadly, fuzzing [71] is a technique for testing a program by automatically generating novel inputs (pseudo-randomly or with some guiding, such as by instrumentation [69]). Fuzzing has traditionally been used to identify inputs that cause a program to crash [71] or to identify security-holes [69].

Execution replay A slightly different space in concurrency analysis is execution replay [90] whereby the program is instrumented to record key events so that the exact execution can be replayed. Execution replay is useful for investigating both bugs and performance issues. They are especially applicable for analysis of bugs and issues that occur in production systems where reproducing the bug may otherwise be impossible[18].

4.5 Discussion

4.5.1 Sleep estimation and full trace-logs

The sleep estimation step of the trace-DAG to trace-log transformation (Section 4.2.4.5) is necessary to generate realistic trace-logs. The step would be unnecessary if the trace-DAG were simplified to a concurrent constraint graph. That is, a graph where all vertices that are not related to concurrent dependencies are removed. However, to retain all function events from the original trace-log, so the estimated trace-log is realistic, these extra steps are required. Realistic trace-logs can account rationally for scheduling overheads and behaviour. Moreover, retaining full information about an execution (functions invoked, function order, etc) provides useful data for analysis, both automatic and potentially manual.

4.5.2 Human refactoring

The changes *Rehype* suggests are designed to be implementable by humans as refactorings that have clear implications (why they are beneficial), as opposed to the effective but hard to understand optimisations a compiler might make. In some cases it can be better to trade a small amount of performance for a cleaner refactoring. For example, in Fig. 4.6 the changed code uses an `CompletableFuture` to wrap and immediately available value so

[18]For example the input that triggered the bug may be sensitive to the user and as such inaccessible.

the return value does not have to be changed (which would require further refactoring). Though this is not optimal for performance (as it involves wrapping the return value in another data structure at runtime), it is easier for a human to implement as it does not require propagating the change through the rest of the codebase and is negligible from a performance perspective (as the cost of wrapping the value in the future is minimal).

Though Chapter 5 presents a method for automatically generating source-code patches, this trade-off is still relevant. Given a generated patch, a developer must (or, at least, should) still understand what the change is doing and determine whether it will have any unintended side-effects. If a change is too complex, the potential for error in the change is much greater (so is the risk that the developer will not use the change due to the complexity).

4.5.3 Is this an artefact of Java's thread implementation?

Theoretically, wait-limited tasks are problematic regardless of the implementation. In practice, runtimes that have lightweight threading systems (such as Go's) are impacted less by this problem than heavyweight thread systems (such as Java) that use operating system threads. If threads are "infinitely" scalable, then using some threads simply to wait is not necessarily problematic. Of course, lightweight threading systems introduce other issues, such as the scheduling overhead inherent in managing thousands of "threads" and potentially losing some CPU pre-emption benefits.

4.5.4 Could better implementation/developer practice avoid these problems?

Yes; as discussed in Section 2.4, perfect use of combinators can result in wait-free task concurrency. However, wait-limited tasks are a real problem that exist in industrial programs today; it seems likely they will continue to be a problem in the future given the relative complexity of using combinators. Fundamentally, more-complex concurrency constructs can achieve better performance, but simpler constructs are more accessible and regularly used by developers. Improving the performance of these simpler constructs (such as by removing wait-limited tasks), can have a significant impact on real-world concurrency usage. In an ideal world, developers would be able to program using simpler constructs and achieve performance close to that achievable with more-complex constructs (or even convert to using the more-complex constructs where useful, as *Rehype*'s convert-to-combinator optimisation does).

4.6 Conclusion

This chapter has introduced *Rehype*, a concurrency performance analysis system that uses execution-trace analysis to propose source-level optimisations and estimate their performance effects. *Rehype* takes a *trace-log* generated by *Quilt* and transforms it into a *trace-DAG*, performs edits to reflect a source optimisation, derives a *trace-log* from the edited *trace-DAG*, and compares the performance-metric values of the estimated *trace-log* against the original. I evaluated *Rehype* on a substantial industrial API server and found the estimation to be highly accurate. Furthermore, improvements suggested by *Rehype* more than doubled the server's potential throughput.

Significantly, the trace-log estimated by *Rehype* for a combination of changes closely matches real trace-logs of executions of the program after implementing the changes, both in nanosecond accurate thread usage (illustrated as concurrent profiles) and in aggregate metrics, demonstrating the estimation method generates accurate and realistic trace-logs.

The key idea of this approach is to estimate *quantifiable* performance effects to identify improvements, instead of detecting *potentially* inefficient patterns. Furthermore, the trace reordering approach can be used to estimate the effect of changes without re-executing the program. While I have focused on task-based concurrency optimisations for Java programs in this chapter, the approach should be applicable to other languages and optimisations.

Future Work

I want to highlight three interesting topics for future work in this area: further per-change analysis, efficient selection of change combinations, and analysing more concurrency models. Beyond these topics there is broad potential for the application of the presented estimation approach for a variety of dynamic analysis tasks such as concurrency bug detection, function-optimisation performance prediction, and general program performance debugging.

Further per-change analysis

Source changes can have positive effects for a particular execution (i.e. analysed trace) while having negative effects for other executions. Currently *Rehype* estimates the effect of changes for a given trace and relies on the developer to assess the broader effects of the changes. Future work could perform further analysis for each change and quantify the trace specificity of a change (similar to over fitting in machine learning), among other per-change analyses, providing more useful suggestions and rankings of changes.

Source-change combination selection

Identifying the optimal combination of changes using a brute-force space search is intractable given the size of the combinatorial space. *Rehype*'s current pseudo-random approach can identify valuable combinations, but is not sophisticated or consistent in finding such combinations. Future work could develop a space search method that is more directed in its approach to combination testing.

More concurrency models

Rehype is currently limited to task-based concurrency using thread-pool scheduling. This constrains its applicability to a subset of potential programs. Future work could implement analysis of other concurrency models and thus expand the set of potential programs, both to those that do not use tasks as well as programs that use a variety of concurrency models. Analysing the parallel use of multiple concurrency models within a single program also presents an interesting challenge.

Chapter 5

Source-Code Patches from Dynamic Analysis

Having identified improvements for a program's concurrency performance using *Rehype*, the next step is to translate the dynamic-data based *improvement specifications* into concrete, static source-code patches. In this chapter I discuss the challenges involved in bridging this dynamic-to-static gap and describe a system, *Scopda*, for generating source-code patches. These are git-diff style patches that can be trivially, and automatically, applied to a codebase, thus completing the capture-analyse-edit cycle. *Scopda* is the final component of *DTRSO*.

Narrative instalment

In our imaginary scenario (Section 1.2.2), Banjo has executed an artificial workload on Paterson, analysed the result with *Rehype*, and now has a list of changes and their estimated effects on Paterson's performance (both thread usage and wall-clock execution duration). Banjo has selected a number of beneficial changes and now needs to implement these changes as small, incremental source-code patches. However, as *Rehype* operates purely in the dynamic domain (see Chapter 4), it defines the changes using their *dynamic contexts* – essentially call-stacks. Determining which piece of code to change based on call-stacks is not obvious, especially in a large codebase. Instead of attempting to do this manually, Banjo uses *Scopda* to generate git-diff-style patches for each of their selected changes. Having run *Scopda*, Banjo can implement the changes as simply as executing the `git` [14] command `git apply change.diff` where `change.diff` is generated by *Scopda*.

Banjo can now push these changes to Paterson's codebase repository as patches for review, along with statistics demonstrating the value of the change (either from *Rehype*'s estimations or by rerunning the artificial workload with the changes applied to Paterson).

This concludes the narrative scenario; Banjo has, using *DTRSO*, analysed Paterson and implemented a number of incremental changes to improve its performance.

5.1 Introduction

Dynamic analysis can identify improvements for programs that cannot be identified using static analysis. In particular, dynamic analysis is well suited to identifying improvements in performance and concurrency, and especially concurrency performance. However,

implementing these improvements is non-trivial as the data they use and emit are not directly connected to the static program structure. This barrier to implementation reduces the utility and adoption of such dynamic analyses.

Recall that *Rehype* generates *improvement specifications*, based on trace data, that can point a developer towards a source-code *change* to be made, but leaves interpreting the specification to implement the *change* to the developer. This interpretation is non-trivial and open to errors.

Scopda automatically generates concrete source-code *patches* for the *improvements* identified by *Rehype* (Section 5.3). A *patch* is the git-style diff between the source code pre- and post-change. *Scopda* maps the dynamic-domain *improvement specifications* into the static domain and then generates source-code *patches* to implement them. While there is a single, generalised method for dynamic-to-static mapping, *Scopda* implements a specialised *change transformation function* (CTF) for each optimisation.

An improvement specification consists of an *optimisation type*, a *caller-path*, and a *callee-tree*. The optimisation type determines the appropriate CTF, while the caller-path and callee-tree determine[1] the specification's *dynamic context*. The dynamic context approximates where the improvement should be made. Caller-paths and callee-trees are structured sets of function *invocations*. For a given invocation, the caller-path is the series of invocations that contain it, and the callee-tree contains the invocations it triggers.

To implement an improvement, the dynamic context must be mapped into a *static location*. A static location is a specific location within the source code (e.g. a function call). In the main example in this chapter (Section 5.2) the mapping is one-to-one, however, in more complex situations a dynamic context may map to multiple static locations (see Section 5.3.2). *Scopda* explores execution paths in the program to map dynamic contexts to static locations. This exploration is performed on an *abstract program graph* (APG), a novel static program representation I introduce that contains inter-procedural control flow and local data flow. After calculating the static locations, *Scopda* performs graph transformations on the APG to implement the improvement and renders the modified APG as source code.

Dynamic-to-static mapping is non-trivial as it has to match sparse dynamic traces (caller-paths and callee-trees) to concrete static locations. The dynamic traces are sparse as *Rehype* only traces a subset of the program's functions (the *tracked* functions) and does not trace sub-function information (e.g. branches). This sparse tracing is necessary to limit the tracing overhead (see Section 3.3).

To enable the exploration of all execution paths in an APG, it must represent the entire program. As many components of a program will be pre-compiled `.jar` libraries (both the Java standard library and third party libraries), *Scopda* supports generating APGs from both Java source code and JVM bytecode. The APG is fundamentally language agnostic and a single APG may contain subgraphs generated from multiple formats. For each supported input format, *Scopda* defines a *language interface* which converts the format to APG subgraphs and, for some formats (i.e. source code but not bytecode), renders APG subgraphs back into the source format.

Section 5.4 explores language features beyond those used in the running example (e.g. conditional control flow, inheritance, and unstructured JVM bytecode). Most features are naturally handled by the semantic *lowering* involved in generating an APG. However,

[1]Intuitively the dynamic context is exactly the caller-path but since the caller-path traces entry points, rather than call sites, information from the callee-tree can refine it.

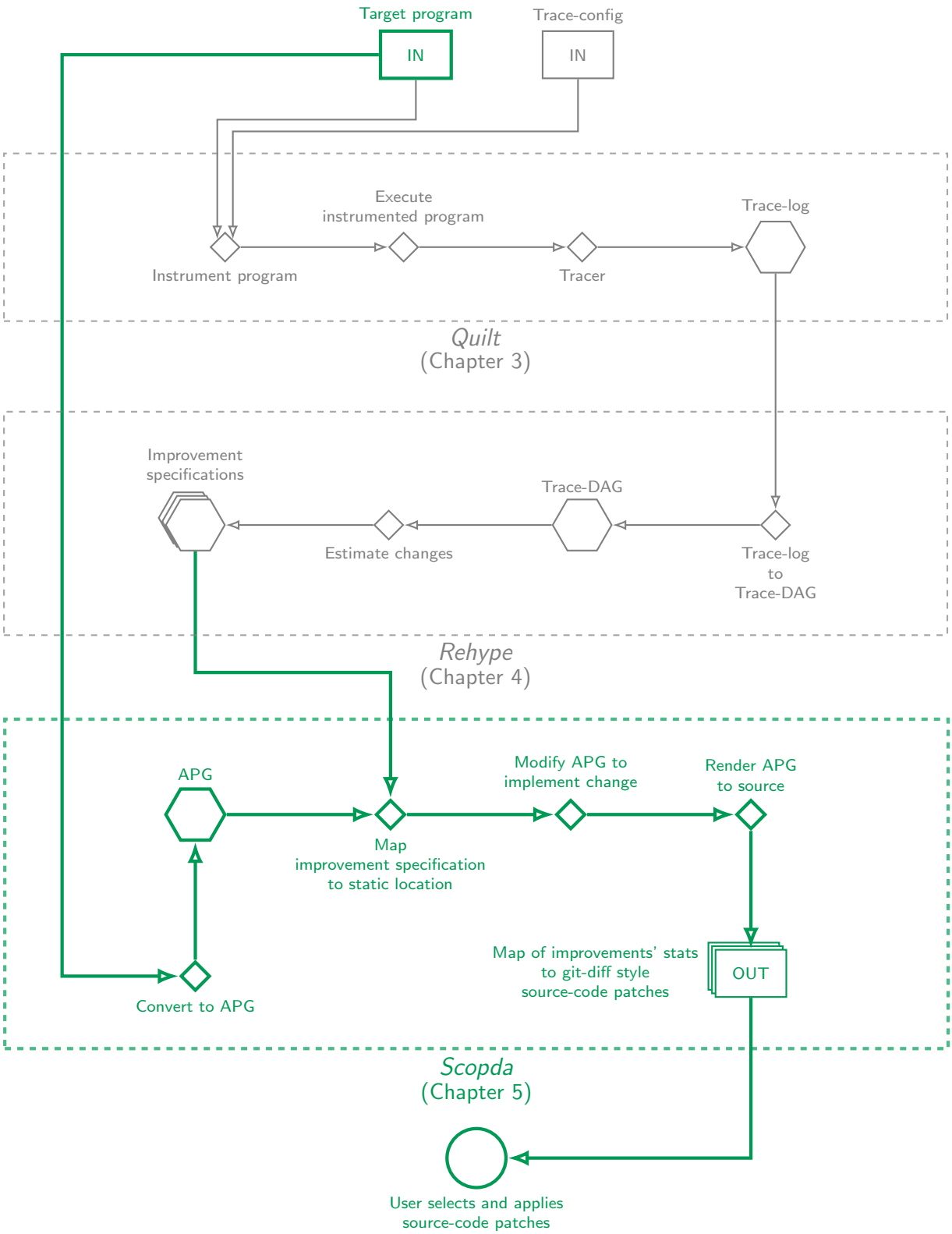


Figure 5.1: High-level overview of where *Scopda* exists within *DTRSO*. *Scopda* converts the target program to an Abstract Program Graph (APG) and uses it to map *improvement specifications*, generated by *Rehype*, to static locations, then modifies the APG, renders it back to source code, and generates a set of git-diff style source-code patches and their corresponding improvement statistics.

some unstructured patterns in JVM bytecode cannot be represented in structured formats such as the APG. *Scopda* handles such patterns at the per-function level using a *grey box* (a simplified representation) approach, whereby precision is sacrificed while preserving safety.

To validate *Scopda*'s approach in the real-world, I apply it to improvement specifications generated by *Rehype* for a large real-world Java program (c. 500kLoC). *Scopda* successfully generates (sensible) source-code patches for each of the nineteen improvements suggested by *Rehype* in Section 4.3.4 (the *Scopda* generated patches are discussed in Section 5.5).

The contributions of this chapter are:

1. A method for generating concrete source-code *patches* based on *improvements* identified using dynamic analysis and *specified* with dynamic-domain data.

2. A method for mapping dynamic traces to nodes in a static code graph.

3. A new static program representation, the *abstract program graph*, that enables: dynamic-to-static mapping, reflecting edits made on the graph back into the original source code, and supporting multiple source formats (e.g. Java source code and JVM bytecode) within a single graph.

Finally, Section 5.6 positions this work among related work, Section 5.7 discusses technical choices and points of interest further, and Section 5.8 concludes the chapter.

5.2 Running example

In this chapter we concentrate on the "convert-to-inline" optimisation – where *Rehype* suggests inlining a spawned task to improve a program's concurrency performance (an example is given in Fig. 5.2) – as it is the most straightforward optimisation, allowing us to focus on the complexities of bridging the dynamic-to-static gap, rather than the complexities of the optimisation. The convert-to-inline optimisation can be applied to certain concurrent tasks to improve resource usage efficiency, see Chapter 4 for more details. *Rehype* specifies the task to be inlined using the caller-path for the task-spawn invocation and the callee-tree of the task execution. The caller-path and callee-tree will not be contiguous on a thread in most instances (i.e. the callee-tree will not be rooted at the end of the caller-path), as the callee-tree will be invoked from a separate thread. *Scopda* identifies the specific task-spawn static location, and derives additional *patch points*, and generates a source-code *patch* encapsulating the improvement. Patch points are source-code locations that are concomitant with the static location (e.g. usages of the `Future` variable the task-spawn function returns). If the task-spawn static location is ambiguous, *Scopda* can:

1. generate multiple patches, one for each static location (the user can choose which to apply);

2. report the ambiguity as an error; and/or

3. generate a new *Rehype trace-config* that, when used for a new program execution, will capture *trace* data sufficient to disambiguate the static location in future uses of *Scopda*.

Fig. 5.2a shows, in JSON format, the improvement specification generated by *Rehype* (simplified[2] for readability), and Fig. 5.2b shows a source-code patch to implement the improvement. In this example `calculateNumber()` contains the task-spawn call and `queryDatabase()` is the *task-body function*. The task framework consists of the spawning function `ExecutorService.submit()` and the root task execution function `Future.run()` (these are the standard task concurrency utilities in Java 8, see the `java.util.concurrent` package documentation [42]). The first invocation in the `calleeTree` is `Future.run()` as it is the first (*tracked*) function called by the thread that executes the task. It, in turn, calls `queryDatabase` via the callback (e.g. a Java `Runnable`) given when spawning the task. Given the improvement specification, *Scopda* identifies (Fig. 5.2b)

- *(static-location)* the `es.submit()` call at line 2 as the task-spawn call to modify;
- *(patch-point)* the *task-body function* `queryDatabase`; and
- *(patch-point)* the `databaseResult.get()` call at line 5 as a use of the task result that should also be modified (as it uses `Future.get()` to wait for the task result).

The modified source-code calls the *task-body function* directly at line 2, updates the returned variable's type, and removes the `Future.get()` call, instead using the variable directly, at line 5.

Determining the benefit of the convert-to-inline optimisation (i.e. identifying it as an improvement) in this example requires dynamic analysis. It is unclear from a purely static perspective whether the elided code (line 3 of Fig. 5.2b) performs significant work or not. In the former case we should retain `queryDatabase` as a task, but in the latter we should inline the spawn as a direct call. By contrast, dynamic analysis can determine how much work the elided code performs and thus whether inlining `queryDatabase` is beneficial.

5.3 Method

Scopda contains three primary components (illustrated in Fig. 5.3): a *language interface*, *dynamic-static mapper*, and *change transformation functions*. The language interface (LI) generates APGs from source-code (and JVM bytecode) and renders APGs back into source-code. The dynamic-static mapper (DSM) takes a dynamic context and an APG and returns the corresponding set of static locations – in effect, nodes in the APG. A change transformation function (CTF) takes an APG and static locations as input and transforms the APG to implement the improvement. One CTF is implemented for each *optimisation*. The appropriate CTF to use is determined by the optimisation type in the improvement specification. The *Scopda* process is thus:

step 1: *(LI)* generate the APG;

step 2: *(DSM)* map dynamic contexts to static locations;

step 3: *(CTF)* transform the APG for the improvement;

step 4: *(LI)* render the transformed APG back into source-code; and

[2] Specifically, method names are not qualified by type signature or containing class, and `interface` names such as `Future` are treated as if they were `class` names.

```
{
  "optimisationType": "convertToInline",
  "callerPath": ["main", "calculateNumber", "ExecutorService.submit"],
  "calleeTree": {
    "function": "Future.run",
    "children": [
      { "function": "queryDatabase", "children": [ ] }
    ]
  }
}
```

(a) A JSON-format *improvement specification*.

```
1: static int calculateNumber(ExecutorService es) {
2:-  Future<Integer> databaseResult = es.submit(
         ThisClass::queryDatabase);
2:+  int databaseResult = queryDatabase();
3:   // ... elided code ...
4:   int otherValue = other();
5:-  return databaseResult.get() + otherValue;
5:+  return databaseResult       + otherValue;
6: }
7:
8: static int queryDatabase() { /* ... */ }
9: static int other() { /* ... */ }
```

(b) Source-code patch generated by *Scopda*.

***Rehype*:** "Execution tracing for the program above has suggested performance gains from: inlining the task that is spawned at the *dynamic context* corresponding to the `"callerPath"` that, when executed, invokes the `"calleeTree"`."
***Scopda*:** "Suggested patch is: calling `queryDatabase` directly from `calculateNumber`, instead of spawning a task for it; concomitantly when calling `queryDatabase` directly, the result is an `int` and does not require a `.get()` call to get the value."

Figure 5.2: Informal explanation of an example "convert-to-inline" *improvement* proposed by *Rehype* (a) and implemented by *Scopda* (b).

step 5: generate a patch (a git-style diff) by comparing the rendered source-code to the original.

5.3.1 Abstract Program Graph

An APG is a unified graph structure containing a program's inter-procedural control flow and local data flow as well as structural (AST-like) information. It has 4 node types (function, variable, operation, and branch) and 15 edge types. It is edge-centric – most semantic details are encoded by the edges. A key property is that it can be rendered back to the unique input AST that generated it, and hence back into the original source-code (modulo white space and spurious bracketing).

The APG contains the semantic information of an AST, its call graph, and its control flow graph. While existing representations, such as graph-overlays [95], contain equivalent information and are equally effective for query and analysis, *Scopda* also needs to modify code for patch generation; the APG's unified design is better suited to transformation as it does not need to coordinate the transformation of multiple overlays.

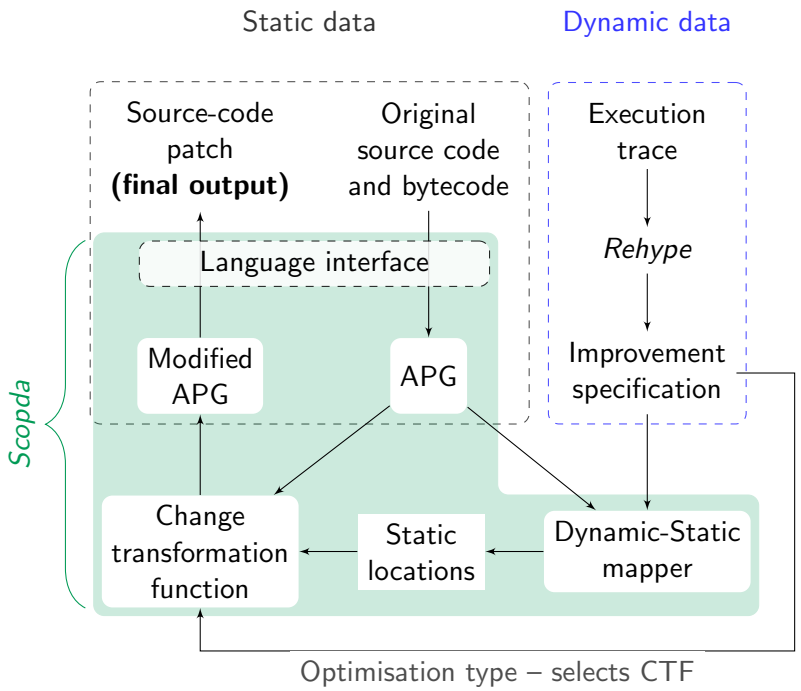


Figure 5.3: High-level structure of *Scopda*. Arrows indicate information flow.

Design principles

The APG design prioritises simplicity of analysis and transformation over simplicity of APG-to-source-code rendering. This priority takes form as a guiding principle: maintain a narrow interface while retaining full expressiveness of (Java) programs. An example of the narrow interface is the simple control flow constructs: standard control flow is described with `branch` nodes and `<goto>` and `<returns>` edges, while exceptional control flow is represented via `<on-exception-goto>` edges (effectively a `catch` statement).

Representation lowering

When generating an APG (*Scopda* **step 1**), a series of *lowering* transformations are applied to normalise source-code representation:

- Local variables are represented in Single Static Assignment (SSA) [97] form (SSA ϕ-functions are represented as independent variable nodes with `<ssa-predecessor>` edges to their SSA *versions*).
- Unnamed variables use A-Normal Form [99] (ANF).
- Non-local variable accesses become calls to *intrinsics*. An intrinsic is a function node representing a language-specific operation, such as accessing an object field or referencing a method. Intrinsics provide a unified approach to various language features and make shared memory access explicit, without affecting the semantics of the program.
- Operators are also converted into intrinsics (e.g. the plus operator in the running example becomes an `Intrinsic::Add` function node in Fig. 5.5).

- All functions are represented statically (i.e. `'this'` parameters are explicit).
- Function and variable symbols are resolved (e.g. `<calls>` edges terminate at function nodes which are also the root of the function implementation graph).

Program behaviour and representation

To understand the APG representation of program behaviour, it is easiest to imagine an interpreter executing the program based on the APG representation. So, for example, when the interpreter executes an operation node, it may take one action (e.g. calling a function) defined by that operation node (which is to say, operation nodes only define a single action per node). We'll use Fig. 5.4, which contains Java code examples and their corresponding APG representations, to understand various behaviours.

Statements and expressions Fig. 5.4a illustrates a simple statement that invokes a function (using the `<calls>` edge), `generateNumber`, and assigns its return value to the variable `x` (using `<generates>`). Here there is one operation node for the statement, it has one action (calling a function), it generates a variable, and it passes no input arguments. Fig. 5.4b is an expression that passes a single argument (using $\texttt{<used-by}_1\texttt{>}$) to a function. Some edges, such as `<used-by>`, use the form $\texttt{<edge}_n\texttt{>}$, where n is a numeric label that defines a local ordering of the edges of that type originating at the same node. For example, if an operation calls a function and passes two arguments, the first argument will be linked with $\texttt{<used-by}_1\texttt{>}$ and the second by $\texttt{<used-by}_2\texttt{>}$. Finally, operation nodes can be nested using the $\texttt{<contains}_n\texttt{>}$ edge. In Fig. 5.4c the result of a call to one function is immediately used as the argument to another function. As noted above, the APG uses ANF, so unnamed values (anonymous variables) are still explicitly denoted with their own variable node. In this example the call to `generateNumber` generates an `anon` variable which is then passed to `consumeNumber`.

Order of operation is determined by the operation node hierarchy. Operation nodes are executed **depth-first** and are ordered according to the $\texttt{<contains}_n\texttt{>}$ edges. So in Fig. 5.4c, when the interpreter reaches the `consumeNumber` operation node, it first continues on to the `generateNumber` operation, executes it first, and then "unwinds" and executes the `consumeNumber` operation. Similarly, in Fig. 5.4d, when the interpreter reaches the first operation node, it continues across the $\texttt{<contains}_1\texttt{>}$ edge first, then across the $\texttt{<contains}_2\texttt{>}$ edge. As noted above, operation nodes may define a single action or no action, in which case they merely act as a structuring tool. The possible actions are:

- Call a function – using the `<calls>` edge.
- Branch through a branch node – `<branches>`.
- Move control – `<go-to>` and `<on-exception-go-to>`.
- Return out of a function (similar to moving control, except passes value(s) along with control) – `<returns>`.

Functions use the `<body>` edge to define a first operation, under which the whole body of the function will be nested. As shown in Fig. 5.4d, each top-level statement in a function becomes an immediate child of the body operation node. Fig. 5.4e illustrates function parameters and function resolution. Function parameters are simply variable nodes linked via a `<param`$_n$`>` edge. As noted above, all functions and variables are fully resolved, which means that any edge terminating at a function (e.g. `<calls>`), terminates at the function node that then defines that function's content. In this example, the two calls to the function `second` (one from `first` and one recursive) are represented by `<calls>` terminating at the `second` function node. Full resolution means that our conceptual interpreter can follow the `<calls>` edge, then the `<body>` edge, and continue executing operations as normal – there is no need for extra context switching or consulting with external lookup tables.

Branching is defined using the branch node and the `<branches>` and `<branch-option`$_n$`>` edges. Fig. 5.4f illustrates a simple `if` statement branching. An important choice in the APG design is for an arbitrary number of branch "options". As the APG is not intended to be executed directly, but rather for analysis and modification, branches do not need to differentiate between "true" and "false" options (or any other form). Instead, analysers simply need to know when the control-flow of a program splits. So, when our conceptual interpreter reaches the operation node that `<branches>` to the branch node, the interpreter crosses both `<branch-option`$_n$`>` edges simultaneously. In practice these branching points act as "mutually exclusive operations" dividers, so an analyser can state that "the program has to take one, and only one, option (control-flow path)".

Loops are implemented as a branching operation containing two branch options, one branch's last operation will have a `<go-to>` edge back to the branch operation and the other branch will have no action/operation (so instead control will simply unwind out of the loop branch operation) – representing the end of the loop. This is illustrated in Fig. 5.4g, where a Java for-loop is illustrated as an APG. For-loops are slightly more complex to represent as they require the loop, an initialising step, a condition check step, and an increment step. This example also illustrates the APG's SSA representation – `<ssa-predecessor>` edges – and how they are represented in recursive structures (in the example, the increment step both uses the SSA ϕ version of the counter variable `i` and generates an SSA predecessor of that ϕ variable).

Exception handling is represented using two edges: `<on-exception-go-to>` and `<caught-param`$_n$`>`. An example is given in Fig. 5.4h. The `<caught-param`$_n$`>` edge works in the same way as the `<param`$_n$`>` edge, except that it originates from an operation node instead of a function node. If an operation is the origin of a `<on-exception-go-to>` edge and an exception occurs during that operation's execution (including execution of the operations contained by it), the control will move to the operation the edge terminates at (similar to how the Java `catch` expression works). If no exception is thrown, then the edge is never crossed. However, the APG does not define a native way for throwing an exception. The APG does not need a throw capability, as it is not executed. Instead, similar to the branching logic, the relevant information for an analyser is that control-flow *could* go to the exception handler at any point during the operation's execution. So, for example using Fig. 5.4h, given a trace-log contained events to the effect "entered, inner, entered except, exited except", an analyser can reconcile the trace-log with the APG as

the APG indicates that the control-flow can switch to the `catch` expression at any point during the `try` expression's execution.

Language-specific annotations

While the APG defines a narrow interface and retains full semantic expressiveness, rendering back into the input AST may require language-specific information. LIs annotate nodes with this language-specific information during APG construction. For example, the Java LI annotates branch nodes to indicate the type of branch statement (e.g. `if`, `for`, `while`). These annotations are opaque to the rest of *Scopda*, but are associated with all copies of a node (CTFs may duplicate nodes).

SSA calculation

Where possible, as with SSA, *Scopda* performs APG construction steps in a language agnostic way (i.e. purely on the APG), allowing various language interfaces to use the same implementation. *Scopda* implements an algorithm based on Braun et al.'s [10] efficient SSA calculation algorithm, though adjusted to operate on the graph format.

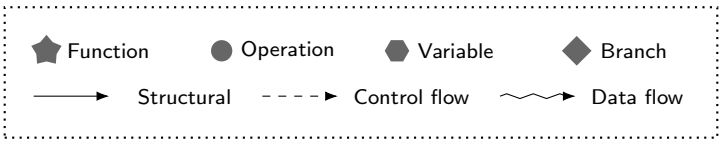
Scopda first constructs a DAG (directed acyclic graph) of basic blocks of operation nodes within a function. Though the basic-block calculation method is APG specific, the fundamentals are the same as other basic-block calculation methods. Each block is annotated with a map, from variable name to variable node, that contains the nodes for named variable generated within the block (this includes variables reassigned within the block as they are represented with new variable nodes). Then blocks are processed in reverse breadth-first order.

Within each block, each operation node's arguments (variables linked via `<used-by$_n$>` connections) are processed to determine if there are multiple possible sources for that variable. If the variable node is generated within the same basic block as the operation node, then there are no other possible sources. However, if the variable node is generated within a previous basic block, then it is possible that a variable with the same name could have been generated in a different basic block and thus require an SSA ϕ variable to be placed in front of them. To find other possible valid variable nodes, *Scopda* traverses back through the basic-block DAG until a variable node with the same name is found for every path through the DAG.

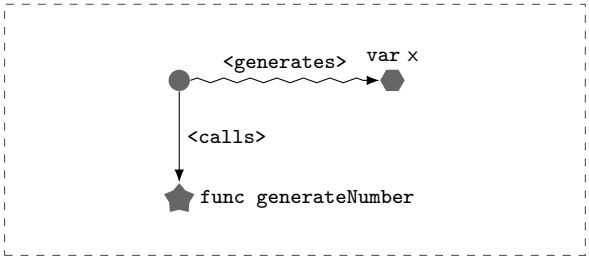
The fact an operation node is linked to a single variable node, while multiple variable nodes may be valid arguments is an artefact of the APG construction process. During construction variable names are mapped to the latest variable nodes to be generated with that name, which means that when an operation node is generated for an expression using that variable name, it is only linked to a single variable node, even if there are multiple possible variable nodes matching that name. This approach is intentional as there are many ways this could occur and it is significant simpler to have a single SSA pass at the end of APG construction, instead of attempting to continuously update the SSA connections throughout APG construction.

Full example

Fig. 5.5 illustrates the APG for the `calculateNumber` method from Fig. 5.2 (pre- and post-change). The operation nodes (circles) resemble an AST statement and expression

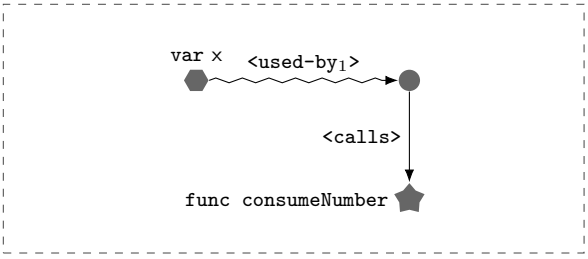



```
int x = generateNumber();
```



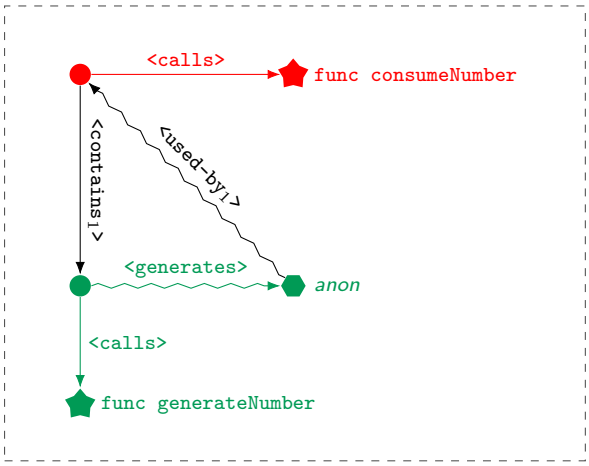
(a) Variable assignment and function invocation example. Variables are assigned using the `<generates>` edge.

```
consumeNumber(x);
```



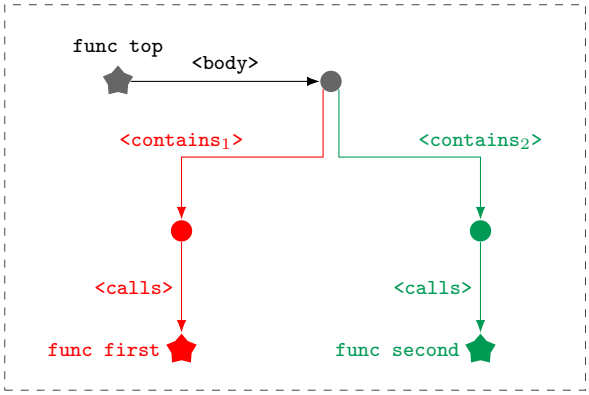
(b) Example of an expression invoking a function with an argument. Arguments to expressions are indicated with a `<used-by`$_n$`>` edge, where n indicates the argument order.

```
consumeNumber(generateNumber());
```



(c) Operation nodes are nested using the `<contains`$_n$`>` edge type, where n indicates the order of operation. In this nested expression, the parent operation (calling `consumeNumber`) contains the child operation (calling `generateNumber`). The child operation generates an anonymous variable which is then used by the parent operation. Recall that operations are "executed" depth-first, so the child "executes" before the parent.

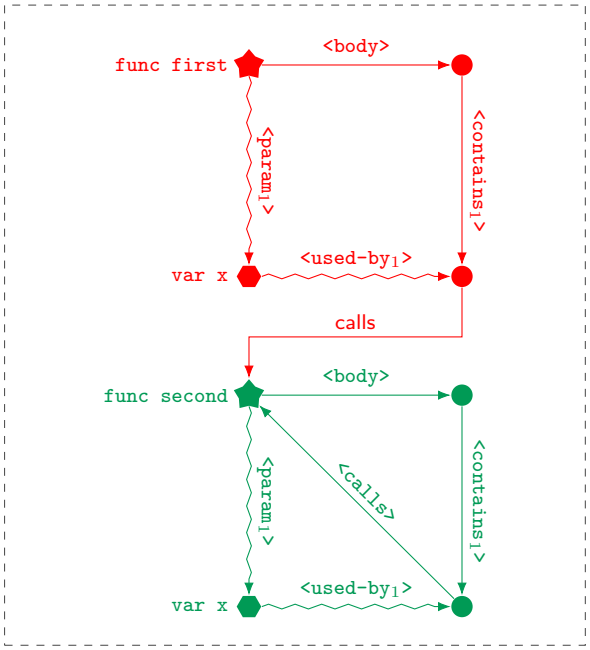
```
static void top() {
  first();
  second();
}
```



(d) Functions have a root "body" operation node, indicated by the `<body>` edge. Blocks of statements are represented by an operation node (such as the body node) that contains multiple nested operations (just as nested expressions do). The order of statements is indicated by the order labels on the `<contains>` edges.

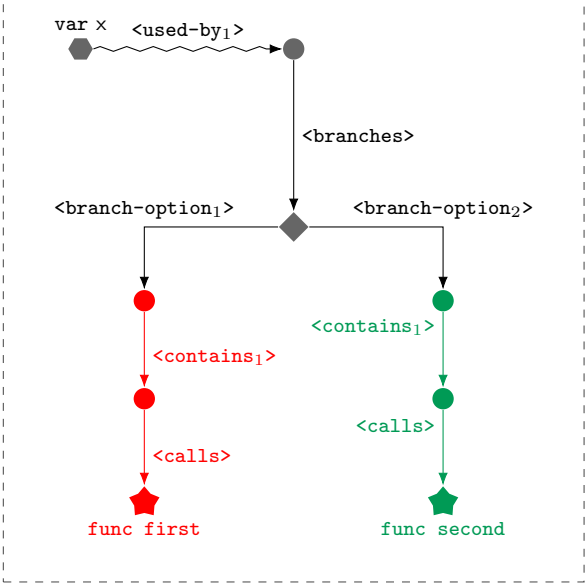
Figure 5.4: Java code examples and their corresponding APG representations. Colours are used purely as visual aides for the reader. All examples assume static functions (i.e. they ignore the `this` argument). *(Cont.)*

```
static void first(int x) {
  second(x);
}
static void second(int x) {
  second(x);
}
```



(e) Function parameters are indicated by `<param`$_n$`>` edges (with n being the order label). Parameters are then used as any other variable, with the `<used-by>` edge. Edges terminating at a function node (e.g. `<calls>`) always terminate on the function node that is then the root of the function definition (i.e. all function usages are fully resolved).

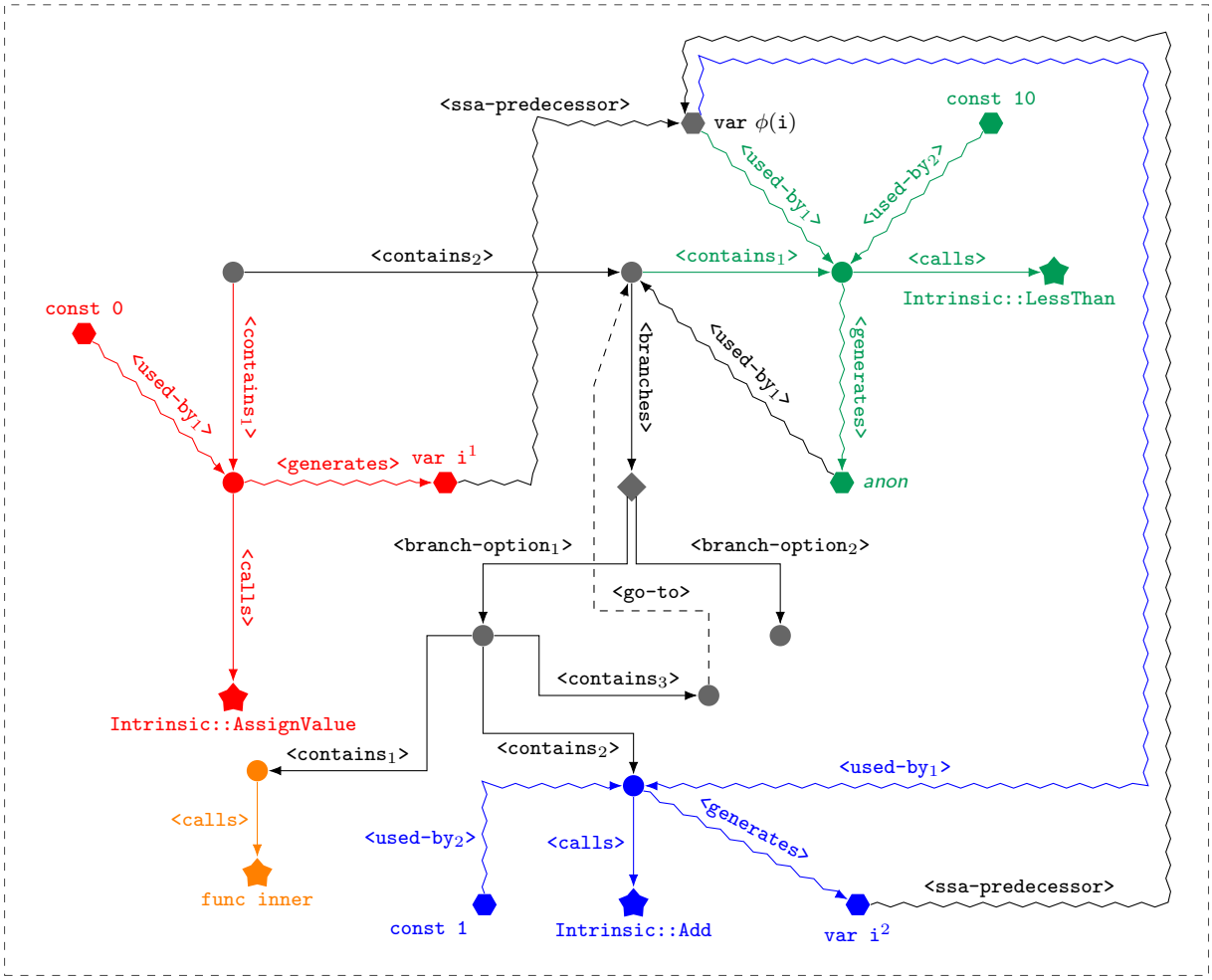
```
if (x) {
  first();
} else {
  second();
}
```



(f) Branching is represented using a `branch` node and the `<branches>` and `<branch-option`$_n$`>` edges. Again, n provides an ordering of branch "options", though the order is only relevant to language interfaces (for APG to source-code rendering).

Figure 5.4: Java code examples and their corresponding APG representations. Colours are used purely as visual aides for the reader. All examples assume static functions (i.e. they ignore the `this` argument). *(Cont.)*

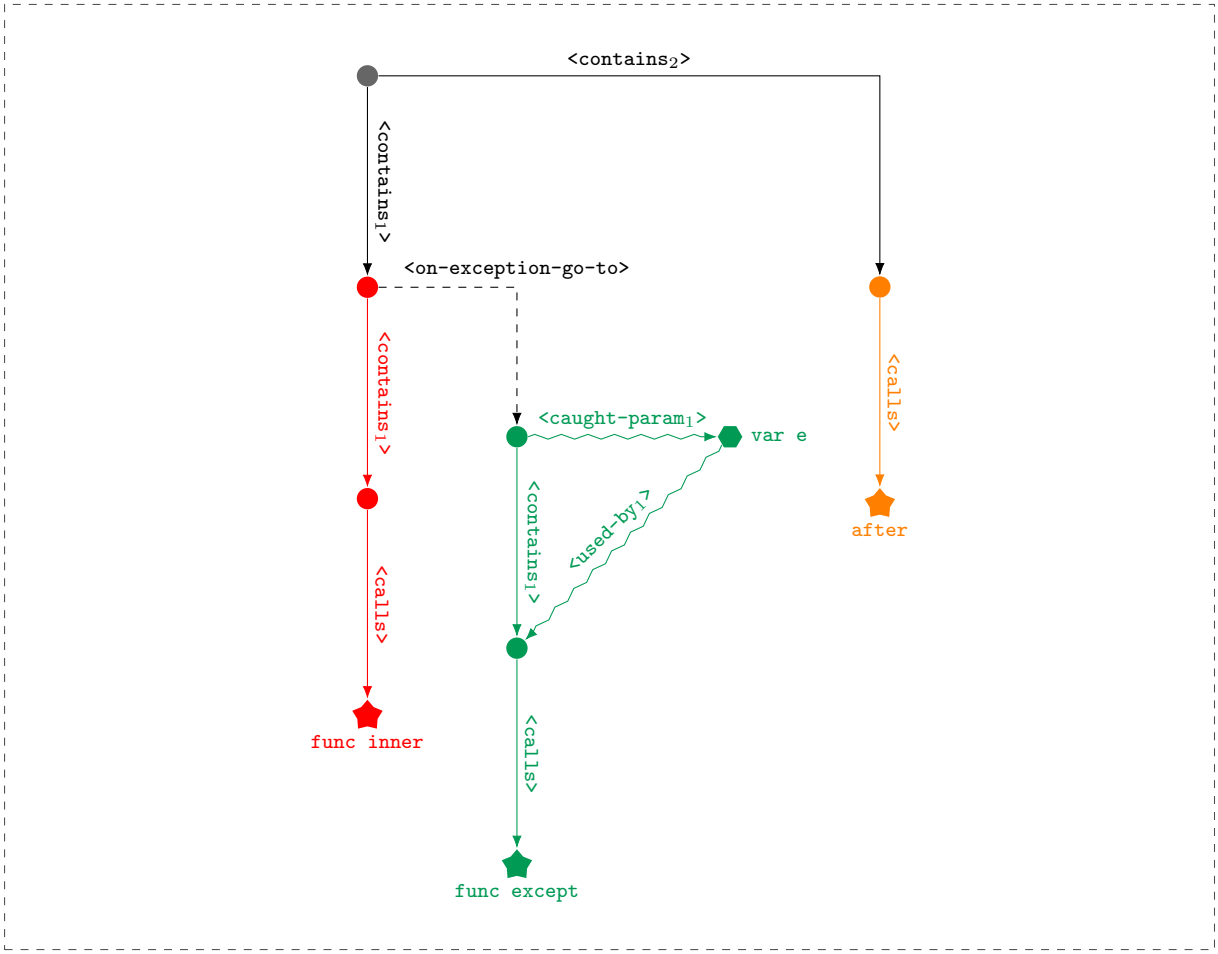
```
for (int i = 0; i < 10; i++) {
   inner();
}
```



(g) Loops are represented with a branch and `<go-to>` control-flow edge (similar to many low- to medium-level representations, such as JVM bytecode). In this example, the root operation node (top-left) first initialises the `i` variable (red nodes and edges), then goes to the branching operation node (top-centre). The branching operation node first generates a boolean condition variable (green nodes and edges) and then branches. As operations are executed depth-first, the condition-generating operation executes before the branching operation executes (by going across the `<branches>` edge). One branch option is the loop body, the other is an empty option (which will result in the operation stack unwinding the depth-first execution). The loop body first performs the execution (orange), then increments the `i` variable (blue), and finally has an operation node that has a `<go-to>` edge back to the branching operation node (causing another iteration of the loop). Finally, in this example we also see that variables use the `<ssa-predecessor>` edge to indicate SSA options. This edge is also applied for recursive SSA options (such as the incremented variable $\texttt{i}^2$ (bottom right, blue)).

Figure 5.4: Java code examples and their corresponding APG representations. Colours are used purely as visual aides for the reader. All examples assume static functions (i.e. they ignore the `this` argument). *(Cont.)*

```
try {
  inner();
} catch (Exception e) {
  except(e);
}
after();
```



(h) Exception handling uses the `<on-exception-go-to>` and `<caught-param$_n$>` edges. The `<on-exception-go-to>` edge is crossed when an exception is thrown while the origin operation node is being executed (e.g. if `inner()` threw an error). If no exception is thrown, the execution proceeds as normal with the depth-first `<contains>` edge-based execution. Finally, the `<caught-param$_n$>` edge acts the same as the `<param$_n$>` edge, except it originates from an operation node instead of a function node.

Figure 5.4: Java code examples and their corresponding APG representations. Colours are used purely as visual aides for the reader. All examples assume static functions (i.e. they ignore the `this` argument).

tree. Though the function implementations are elided, the `<calls>` edges connect directly to the target function nodes, providing the call-graph information. The operation and branch nodes along with the solid (structural) and dashed (control flow) edges define the control flow. The variable nodes and zigzag edges define the data flow.

5.3.2 Dynamic-Static Mapper

The dynamic-static mapper converts a given dynamic context into a set of static locations (*Scopda* **step 2**) by exploring static execution paths in the APG. To identify the static locations that correspond to a dynamic context, the mapper identifies all possible *static code paths* (SCPs) that can correspond to a caller-path (this method generalises to callee-trees as a tree can be treated as a series of paths). An SCP is a series of nodes in the APG connected by control flow edges. An SCP is *valid* for a caller-path if the SCP, filtered to *tracked* function nodes, is equal to the caller-path. Each node within an SCP is a static location. Once the SCPs are calculated, the CTF for the optimisation selects relevant static location(s) from the SCPs. For example, the key static location in my running example (Fig. 5.2) is the operation node for the call to `ExecutorService.submit` (highlighted in the topmost dashed (blue) circle in Fig. 5.5).

In simple cases, including Fig. 5.2, every dynamic context maps to a single static location. In richer situations, such as when a function can be called from two different dynamic contexts, a dynamic context may map to multiple static locations.

Scopda's execution path exploration is intuitively a guided graph traversal along APG control flow paths. It identifies SCPs that are *valid* for a given caller-path by testing all control flow paths starting at the first invoked function in the caller-path and terminating at the last invocation in the caller-path. A control flow path is invalidated (and exploration on it is terminated early) when it encounters a *tracked* function node that does not correspond to the next invoked function in the caller-path.

A naive approach might generate a set of concrete SCPs using standard traversal, but this would not scale to real-world programs. Such an approach would result in a combinatorial explosion of SCPs when there are many possible static code paths between *tracked* functions (in fact, there may be infinite paths given recursion). Moreover, the number of possible paths makes it computationally infeasible to sequentially iterate over them.

Instead, when given a caller-path, *Scopda* uses the call graph to produce a caller-path-enhanced call graph. The paths in this caller-path-enhanced call graph are exactly the set of valid SCPs. In this graph, invocations, that are adjacent in the caller-path, are interposed by a graph representing the SCPs of non-*tracked* functions between the invoked functions. This is illustrated in Fig. 5.6.

More concretely, given a caller-path p and an invocation p_i within it, and writing G for the program's call graph, define $G^{\text{forw}}(p_i)$ to be the subgraph of G which can be reached from p_i without passing through a *tracked* function node. Similarly define $G^{\text{back}}(p_{i+1})$ as the subgraph of G which is backwards reachable from p_{i+1} without passing through a *tracked* function node. Then, the graph of SCPs that interposes two adjacent invocations, p_i and p_{i+1}, is calculated as $G^{\text{forw}}(p_i) \cap G^{\text{back}}(p_{i+1})$ – the non-*tracked* sub-call-graph that is reached from p_i and can reach p_{i+1}.

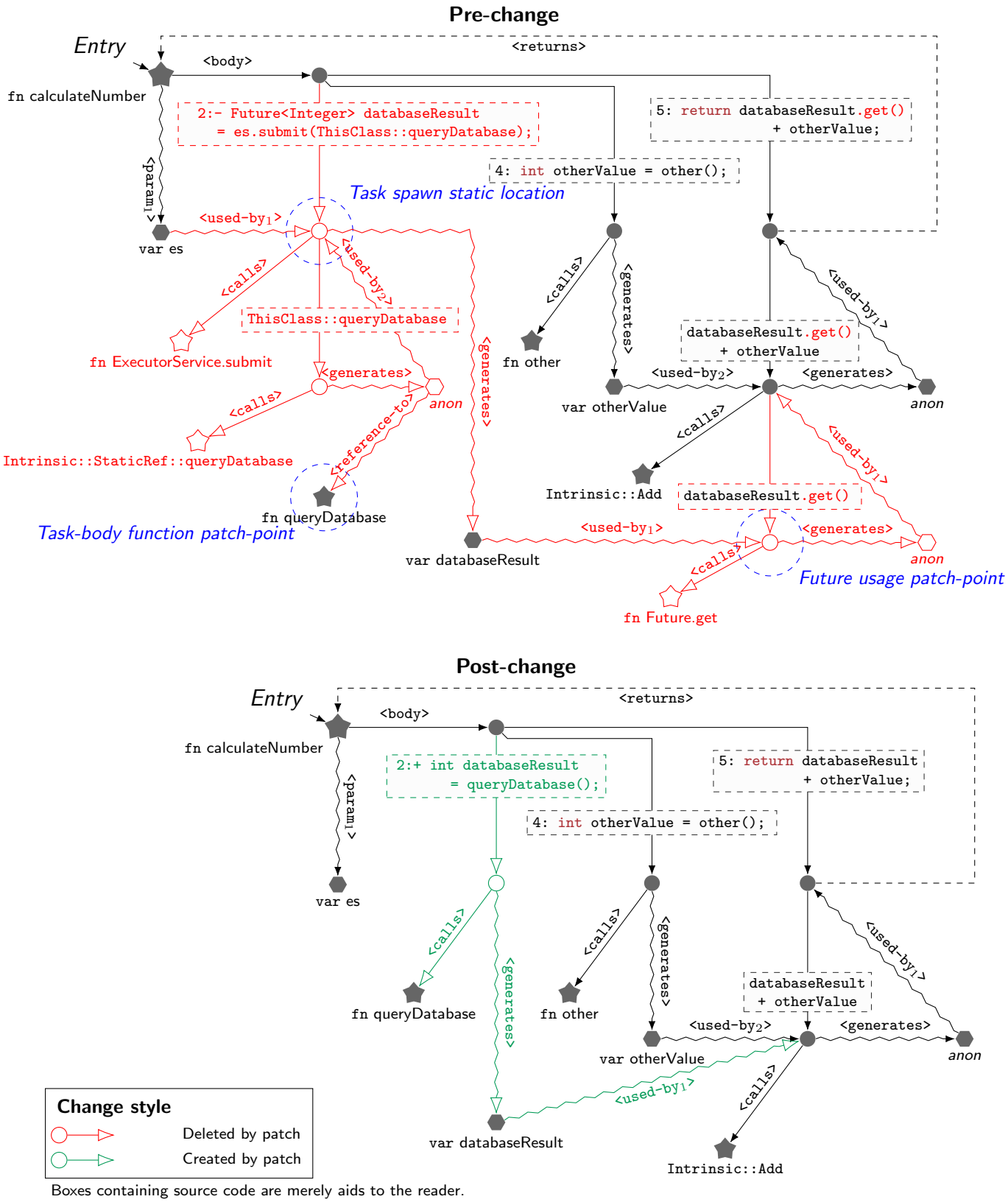


Figure 5.5: APG representations of `calculateNumber` before and after the patch from Fig. 5.2b. Hollow nodes and edge arrowheads are those deleted or created by the patch (these also follow git-diff colour conventions).

Tracked functions: $\{f, k, j\}$ *Caller path:* (f, j)

$$G^{back}(j) = \{b, g, d, h, y, q\} \qquad G^{forw}(f) = \{b, g, d, h, x, p\}$$

$$G^{forw}(f) \cap G^{back}(j) = \{b, g, d, h\}$$

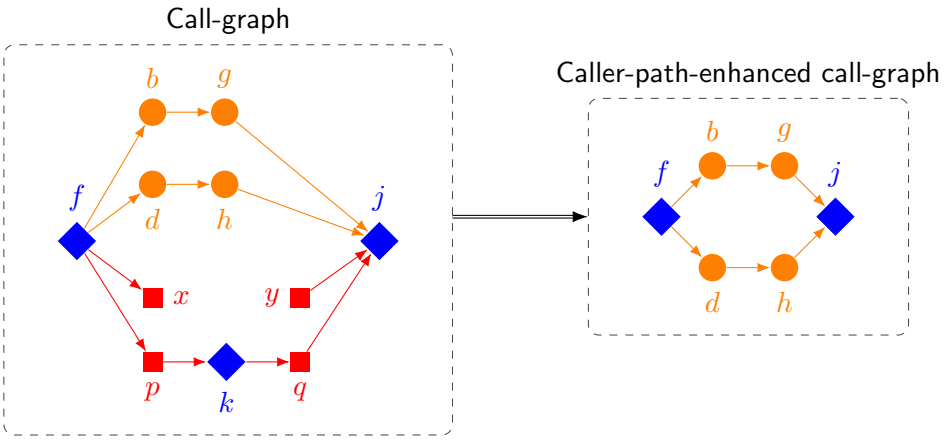


Figure 5.6: Example of a *caller-path*-enhanced call-graph calculated for a given call-graph, caller-path, and set of tracked functions. The blue diamonds are tracked functions (i.e. they will appear as invocations in the caller-path if executed during tracing), the orange circles are untracked functions that are in the reachability sets ($G^{forw}(f)$ and $G^{back}(j)$) and are thus included in the caller-path-enhanced call-graph, and the red squares are untracked functions that are in one but not both of the reachability sets.

5.3.3 Change Transformation Functions

Scopda's CTFs implement improvements (*Scopda* **step 3**) by transforming the APG (e.g. by inlining a task's execution). CTFs take an APG and a set of static locations as input, and generate a transformed APG. CTFs are implemented as a series of graph analyses and transformations applied to the original APG.

For example, in the running example (Fig. 5.2), the *convert-to-inline* CTF's process is:

1. Derive additional *patch points* (Section 5.2) from the task spawn static location (the `submit()` call) by identifying:
   (a) The second argument to the `submit()` call (the `<used-by$_2$>` edge terminating at the task spawn static location in Fig. 5.5) is the *task-body function*.
   (b) The `Future.get()` call, that takes the `submit()` call's result variable (`databaseResult`) as the first argument, must be removed.
2. Modify the `submit()` call to call the task-body function (Fig. 5.7a).
3. Make the result variable, `databaseResult`, an `int` instead of a `Future<Integer>`.
4. Remove the call to `Future.get()` and replace it with a simple usage of the `databaseResult` variable (Fig. 5.7b).

Steps 2 and 4 can be represented in our graph grammar (recall the grammar from Section 4.2.4.3) as follows. To modify the `submit()` call, define a function of the task submission operation x as:

$$CallDirect(x) = \begin{matrix} \texttt{<used-by}_2\texttt{>}(y, x) \\ \texttt{<reference-to>}(y, f) \\ \texttt{<calls>}(x, \texttt{submit}) \end{matrix} \mapsto \texttt{<calls>}(x, f) \tag{5.1}$$

Here, the task-body function is node f.

Similarly, to replace the `Future.get` call with a simple usage of `databaseResult`, define a function of the task submission operation x as:

$$ReplaceUsages(x) = \begin{matrix} \texttt{<generates>}(x, g) \\ \texttt{<used-by}_1\texttt{>}(g, w) \\ \texttt{<calls>}(w, \mathit{Future.get}) \\ \texttt{<generates>}(w, v) \\ \texttt{<used-by}_n\texttt{>}(v, u) \end{matrix} \mapsto \begin{matrix} \texttt{<generates>}(x, g) \\ \texttt{<used-by}_n\texttt{>}(g, u) \end{matrix} \tag{5.2}$$

Here, the `databaseResult` variable would be node g. Note that the parameter usage index, n, is preserved in the edit.

These graph edits are illustrated in Fig. 5.7, while the overall result is illustrated in Fig. 5.5, where the top and bottom graphs are pre- and post-change versions, respectively. Edges changed between the versions (corresponding to the patch) have enlarged arrow heads and nodes created/deleted are hollow. Modified edges and nodes also follow git-diff colour conventions.

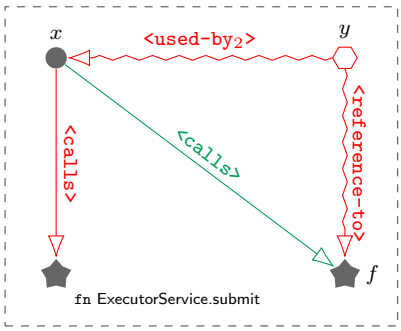
5.3.4 Rendering Source Code

There are two subtleties in the APG-to-source-code rendering process (*Scopda* **step 4**). First, APG to AST conversion is done per-function; higher-level constructs, such as classes, are rendered unchanged (save for the modified functions). Since *Scopda* operates at the function level, for both analysis and transformation, no other source code needs to be changed. Second, the final patch is minimised at each output step (APG-to-AST and AST-to-source-code); this is important for adoption as larger and more complex patches are harder for developers to check.

At each output step, *Scopda* copies as much of the original data (AST nodes or source code text) as possible and only generates new versions for the parts that have been modified. At the APG to AST step, *Scopda* copies the original AST for the function being rendered and replaces only those nodes that correspond to modified APG subgraphs. Similarly, at the AST-to-source code step, if an AST node is unmodified the corresponding source code is copied, whereas if it is modified the source code is generated from the AST. This process of using the original data where possible ensures that the resulting diff is minimal (and non-functional aspects of the code, such as code style and comments, are retained).

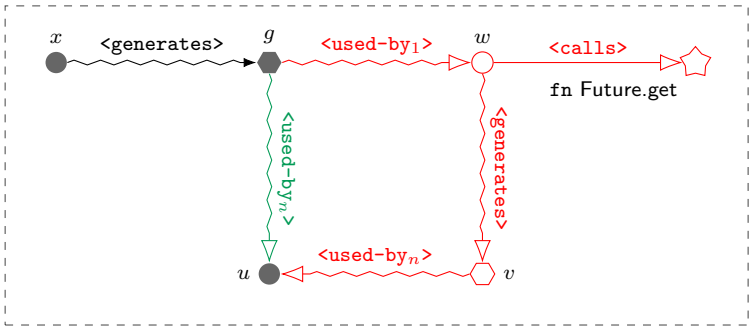
Finally, the output patch (*Scopda* **step 5**) is generated by applying git's diff algorithm to the original source code and generated source code.

$$\begin{aligned} &\mathit{CallDirect}(x) = \\ &\texttt{<used-by}_2\texttt{>}(y, x) \\ &\texttt{<reference-to>}(y, f) \mapsto \texttt{<calls>}(x, f) \\ &\texttt{<calls>}(x, \texttt{submit}) \end{aligned}$$



(a) Step 2 of the CTF process (Eq. 5.1), replace the `submit()` call with a call directly to the task body function.

$$\begin{aligned} &\mathit{ReplaceUsages}(x) = \\ &\texttt{<generates>}(x, g) \\ &\texttt{<used-by}_1\texttt{>}(g, w) \\ &\texttt{<calls>}(w, \mathit{Future.get}) \mapsto \\ &\texttt{<generates>}(w, v) \\ &\texttt{<used-by}_n\texttt{>}(v, u) \end{aligned} \quad \begin{aligned} &\texttt{<generates>}(x, g) \\ &\texttt{<used-by}_n\texttt{>}(g, u) \end{aligned}$$



(b) Step 4 of the CTF process (Eq. 5.2), remove the `Future.get()` call and replace it with a simple usage of the variable generated by the updated submit operation.

Figure 5.7: APG illustrations of the graph edits performed by the *convert-to-inline* CTF. The APG legend is the same as in Fig 5.5.

5.4 Real-world complexity

The running example does not include various features present in real-world programs, such as branch statements, inheritance, unavailable source code, and unanticipated binary code structures. I now summarise how the APG, and *Scopda* more generally, handles these, to illustrate *Scopda*'s approach to real-world complexity.

Branching While branching statements can generate various branch orders and behaviours (e.g. `if` vs. `switch` statements), such distinctions are irrelevant to the analysis and transformation performed by *Scopda*. The dynamic-static mapper explores all branch options irrespective of order. Similarly, CTFs that modify branching statements consider the branches as identified by the *static locations*, not their order. Therefore, the APG identifies branches as distinct from each other, but does not specify *how* they are distinguished by the source language (LI annotations can indicate this for source-code rendering).

Inheritance and interfaces Though omitted in the running example, the APG supports inheritance with an `<overrides>` edge linking a function implementation to an interface function it inherits/overrides. In analysis these edges are, effectively, expanded as `<calls>` edges between operation nodes that call the interface function node and the implementation function nodes. *Scopda* does not (yet) perform call-graph reduction using infeasible path analysis [115] as the dynamic-static mapper's reachability-based approach for constructing graphs of SCPs is currently sufficient for the analyses it performs.

Source-code availability APGs must contain all *tracked* functions and all functions on static call paths between *tracked* functions to enable dynamic-to-static mapping. In

many cases this includes functions for which the source code is unavailable (e.g. third-party libraries). As such, *Scopda* also implements a *language interface* for generating APGs from JVM bytecode, though it cannot render bytecode derived nodes into Java source code. Individual APGs may contain subgraphs derived from multiple input formats/languages (e.g. JVM bytecode and Java source code).

JVM bytecode and grey boxes JVM bytecode is an unstructured format in which there are valid bytecode patterns[3] that cannot be represented in structured graphs (Miecznikowski and Hendren [70]), such as an APG. These patterns are addressed by an additional APG construct, the *grey box*. Grey boxes contain only those nodes and edges to enable inter-procedural analysis (e.g. call-graph edges) by the dynamic-static mapper; they do not support code modification. Grey boxes provide safe, but imprecise, information. Moreover, grey boxes permit *Scopda* to follow a *fall back to simpler representation* principle when encountering unanticipated structures in code; this enables support for unstructured formats and eases support for new languages. While grey boxes have the potential to reduce analysis precision, they should not impede final patch generation (and do not in my experimental experience (Section 5.5)) as they never represent source-available functions (i.e. code which patches affect).

```
// Track main(), g(), and h(), but not untracked1() nor untracked2().
void main() {
  if (cond) {
    untracked1();
  } else {
    untracked2();
  }
}
void untracked1() { g(); h(); }
void untracked2() { h(); g(); }
```

Figure 5.8: Representing `untracked1` and `untracked2` as *grey boxes* causes ambiguity in *Scopda*'s analysis.

As an example of the analysis imprecision that grey boxes can cause, Fig. 5.8 presents a constructed case that, with grey boxes, could cause ambiguity in *Scopda*'s analysis. In the example, functions `main`, `g`, and `h` are tracked and functions `untracked1` and `untracked2` are not. The `main` function calls `untracked1` and `untracked2`, which both call `g` and `h` but in different orders. Given a trace-log showing `main` calling `g` and then `h` in turn, *Scopda* could determine that `untracked1` was called (due to the order of the `g` and `h` invocations). However, if `untracked1` and `untracked2` were both grey boxes, *Scopda* would not be able to determine which one was called as both would indicate that they call `g` and `h`, but would not indicate the order. If a patch then depended on which was called (e.g. a patch that needs to edit `main` in some way), this ambiguity could impede patch generation.

Implementation In practice it is faster to generate APGs from JVM bytecode instead of raw `.java` files as it does not require further compilation (raw `.java` files must be compiled by `javac` to resolve types for the APG, whereas these are already resolved in

[3]Note that these unrepresentable patterns are **structural** patterns, such as multi-entry basic blocks, not **behavioural** patterns that enable dynamic behaviour, such as Android Intents. Behavioural patterns can be represented in the APG format just as any in any other structured format (e.g. Java). Fundamentally, any structural pattern that can be represented in a structured programming language can be represented as an APG.

bytecode). As the APG representations of bytecode and source-code versions of a function are semantically equivalent, I initially generate the entire program APG from bytecode and only generate APG function representations from source-code when necessary. Specifically, when source-code is available for a function *Scopda* uses it if

- the bytecode version cannot be successfully generated (i.e. it would be a grey box); or
- the CTF must transform the function to generate a patch.

If the CTF attempts to transform a source-unavailable function, an error is returned.

While this approach requires generating APG subgraphs twice for functions to be edited (first from bytecode and then from source code), potentially presenting scalability issues for large code bases, in practice this is not a significant issue. APG generation is constant w.r.t. the number of `.java` files to be edited for a given set of patches. Only those `.java` files that will be edited in at least one patch are converted into an APG, the rest of the source code can be ignored (the vast majority of code in large projects). Ignoring non-patch-relevant code during (`javac`) compilation is possible as the compiled `.jar` file – which is executed to generate the original trace and is also used as the basis of initial APG construction – can be given as a class-path library to the `javac` compiler, providing all dependencies required by the `.java` files. Naturally, if a `.java` file is affected by multiple patches, it only needs to be compiled once as the same APG can be used to generate multiple patches.

5.5 Application to real-world

As with *Quilt* and *Rehype*, I apply *Scopda* to *Acme* (Section 2.6.1), a (proprietary) industrial Java API server (c. 500kLoC) for a consumer web and mobile application. *Scopda* successfully generates source-code patches for each of the nineteen *improvements* suggested by *Rehype* (the same improvements described in Section 4.3.4). Fig. 5.9 contains one of the real patches generated by *Scopda*.

The APG is constructed from the server's source code `.java` files, compiled `.jar` file, and the Java 8 SE standard library `.jar` file. The final APG contains 8 690 403 total nodes, 409 231 functions, and 223 492 intrinsics. Of these, the JVM bytecode *language interface* successfully generates APG representations for 398 699 functions, 97.43% of all functions. *Grey box* versions are generated for the remaining functions (all are in third-party `.jar` files where source is unavailable).

The server with patches automatically applied, by `git apply`, successfully executes and exhibits the predicted performance increase (see Section 4.3).

Recall that, in this thesis, we are interested in the theoretical development of a practical technique. As such, the interesting aspect of this evaluation of *Scopda* is that it works when applied to a real-world codebase, using real-world *Rehype* improvement specifications, not how fast it works (though it is fast, in the order of seconds) or how many improvement specifications it is unable to handle (though it is able to handle all generated by *Rehype* for *Acme*). Of course, there are instances where *Scopda* will not be able to handle an improvement specification, as discussed above. However, such points are more interesting for discussion and theoretical consideration, than for quantitative evaluation.

```
public UncheckedFuture<ByteBuffer> sign(ByteBuffer value, Key key) {
  return executorService.submit(() -> {
    try {
      SecretKey secretKey = getKey(key, true);
      Mac mac = Mac.getInstance(secretKey.getAlgorithm());
      mac.init(secretKey);
      return ByteBuffer.wrap(mac.doFinal(value.array()));
    } catch (NoSuchAlgorithmException | InvalidKeyException e) {
      throw new RuntimeException(e);
    }
  });
}
```

Figure 5.9: An example patch generated by *Scopda* for a real-world Java server.

5.6 Related work

I am unaware of existing work that modifies static source code based on dynamic analysis in a similar manner to *Rehype* and *Scopda*. Though, there are examples of static-analysis optimisation being performed based on dynamic analysis, such as JIT compilers deciding whether to compile an interpreted function to native code based on its performance at runtime. Such JIT compilation utilises dynamic analysis to decide whether to compile a function (though the "analysis" may be as simple as a threshold test on the number of invocations and cost per-invocation) and static analysis in optimising the compiled native code. This differs from *Rehype* and *Scopda* in that it does not modify the original static source code (the source code file does not change), it is, instead, an ephemeral compilation of the static code.

Work relevant to the APG includes established general representations such as control-flow graphs [3] and call graphs, system-specific internal representations (e.g. LLVM's IR [54]), and task-specific representations such as source-code query graphs [95, 83], among others [108]. These differ from the APG in specificity (e.g. CFGs), target use (query graphs), or structure (compiler IRs designed for optimisation and machine-code generation). By nature of the target application, the APG does not need to represent some information that is critical to other IRs, such as an efficiently checkable type-system, as it can assume the input programs are valid. Within the context of compiler IRs, the APG uses a form of semantic lowering that can be uniquely reversed (i.e. the lowered IR can generate the unique input high-level source code (modulo white space and spurious bracketing)), whereas most IRs do not require this attribute [18]. Furthermore, given its unified approach (as opposed to, for example, overlay graphs), the APG is particularly well suited to algorithms that combine analysis and transformation.

Some existing work combines static analysis and dynamic analysis for the detection of bugs [4, 127], especially multi-threading bugs which are particularly difficult to find with static analysis alone. Such work combines static analysis and dynamic analysis to enhance the analysis (detection) process, whereas *Scopda* combines them to translate dynamic analysis identified *improvements* back into the static domain (source code).

5.7 Discussion

Scopda takes a somewhat unusual static analysis approach given its focus on bridging the

dynamic-to-static gap. The design decisions regarding the APG encapsulate many of these oddities. For example, it represents branching statements, but does not represent which branch is true, false, or other, as the analysis does not need to. Similarly, it has a native representation for exception handling, but not exception throwing. In this section we will consider various oddities of the APG and broader *Scopda* design, and what lessons we might take from them for other applications.

We begin by considering a key principle of the APG design, that it is to be used to analyse valid execution paths, not to be executed itself. Then we will look at a key design differentiator between the APG and other representations such as ASTs, the APG uses composition not inheritance. Finally, we will broaden out and consider the benefits of designing specialised data structures (or representations) for analysis, as opposed to using generalised versions.

Valid execution path analysis, not executable

Scopda's primary analysis phase is the dynamic-to-static mapping process, during which a dynamic trace needs to be mapped to static code paths (SCPs). This involves checking a recorded sparse trace against the static APG to identify which paths in the APG could have produced the trace, and thus which static paths may have been executed.

Though superficially similar to executing a program, as it involves following control-flow through the program, it is significantly different as it does not evaluate data to decide whether a control-flow option is executed or not (e.g. which branch is executed). Instead, it, conceptually, considers all possible control-flow paths and eliminates those that are unreconcilable with the trace.

This means the APG design only needs to represent possible paths, and when paths branch, but does not need to represent *how* those paths branch (e.g. which branch option is a true/false option in an if-statement).

Composition and disentangling semantics

The APG's semantic representation can be thought of as composition to ASTs' inheritance, in a similar vein to type system composition versus inheritance. Where ASTs have specific (nested) node types (e.g. `ExpressionNode`, `ArithmeticExpression`, `PlusExpression`) the APG has a single operation node type and multiple edges that define the semantics, but the edges do not depend on each other – they are composed. This approach separates (or disentangles, if you will) the semantics of an expression into discrete edges, rather than being implicit in an AST node type.

Intrinsics take this a step further by representing language-specific features – irrelevant to the change transformation functions but important to the language interfaces – as callable functions. This compositional approach allows analysis and transformation algorithms to deal directly with the semantics shared by different expressions and relevant to the algorithm, rather than handling each node type individually. Importantly, this reduces the complexity burden on transformation algorithms to maintain a valid APG.

Specialised vs. generalised representations

While there are many generalised static code representations, such as CFGs [3], ASTs, and call-graphs, designing a specialised representation (e.g. LLVM's IR [54]) can significantly

reduce the complexity of analysers built on it. Specialised data structures allow us to encode exactly the information required, in the most convenient form. They also enable operations that might be impossible, or very difficult, in generalised structures. In essence, using a specialised representation moves complexity from the logic layer (the analyser) to the data layer (the representation), the specialised representation encodes the relevant complexity instead of the analyser encoding the complexity as logic combining existing representations.

Two examples illustrate this well, representation lowering and maintaining representation integrity during transformation. Many representations, including the APG (Section 5.3.1) and LLVM's IR [54], perform some form of lowering – reducing the level of abstraction – during construction. In fact, a standard pattern found in compilers is the use of "phases", each of which may perform some type of lowering [22], until the final executable code[4] is produced (conceptually, a compiler can be thought of as a series of lowering steps between the high-level source code and the lower-level executable code). However, the APG differs to many other representations in its ability to reverse this, "raising" the level of abstraction when rendering back into source code. Similarly, many general purpose representations aim to represent only one aspect of the program (e.g. ASTs represent code structure, CFGs represent the control-flow). This means that analysers, which analyse multiple aspects of a program, must use these representations in concert with each other (hence the development of utilities such as graph-overlays by Rodriguez-Prieto et al. [95]). However, when transforming the program's source code, as *Scopda* does, the code representation integrity must be maintained so the generated patches are valid, a particularly difficult prospect when coordinating multiple representations.

This is a fundamental drawback of generalised representations. Given the definition of a specialised representation as being designed for a specific analysis, then a "generalised" representation is naturally one designed without a specific analysis in mind. It is unlikely, though theoretically not impossible, that such a generalised representation will be able to represent all aspects of a program in a usable form[5]. If we accept that such an all encompassing representation does not exist, then all generalised representations must each represent a subset of relevant aspects (likely a single "aspect" per representation).

Any analyser that involves multiple aspects of a program will, naturally, have to use either a composite of generalised representations or a specialised representation. If the analyser involves transformation, then it must maintain integrity of the representation(s) it uses. When using a composite representation, this requires maintaining integrity *between* representations, which is a challenge.

An analyser could, of course, use generalised representations, perform transformations, and maintain the integrity of the representations. Though, doing so would significantly increase the complexity of the analyser. Conceptually it would, essentially, be creating a specialised representation by binding the generalised representations together using a logic layer. While possible, such an approach seems to be more work than designing a specialised representation.

For further reading, Cooper & Torczon discuss code representations, in the context of compilers, and their competing objectives in their book *Engineering a Compiler* [22],

[4]The form of the executable code will differ based on the runtime. For example, Java programs will be compiled to JVM Bytecode which is then executed by the JVM, whereas C programs will normally be compiled into an assembly language (e.g. x86 assembly).

[5]If achieved, this would be "the one representation to rule them all".

Chapter 5. Aho et al. also discuss compiler representations and static analysis methods (such as interprocedural analysis) in their book *Compilers: principles, techniques, & tools* [1].

5.8 Conclusion

This chapter described *Scopda*, a system for generating source-code *patches* for *improvements* identified by *Rehype*'s execution-trace-based dynamic analysis. *Scopda* first maps the dynamic-domain *improvement specifications* into the static domain using a general method and then applies a code transformation method specialised to each optimisation. *Scopda* uses a custom static program representation (abstract program graph, APG) to map between dynamic and static domains and perform code transformation within a single representation. Applying *Scopda* to *Acme*, a large real-world program, demonstrated that it generates sensible source-code patches in the real-world.

Scopda bridges the dynamic-to-static gap to enable dynamic analysis to be translated into practical static changes. Though a number of tools cross the static-to-dynamic gap, going the other direction is non-trivial. In particular, as dynamic analysis is naturally based on limited knowledge of the target program (e.g. sparse tracing, Section 3.3), determining the relevant static components is a challenge.

The APG representation is a core enabler of *Scopda*'s approach. Most importantly, it enables the analysis *and* transformation of the program on a single representation. This is possible, without significant extra algorithmic complexity in the analyser, as the representation is specially designed for *Scopda*. I have argued that designing specialised representations provides a number of substantial advantages, particularly for analysers that involve transformations.

In the future I plan to develop support for more optimisations and expand *Rehype* and *Scopda* to work on more programming languages. I also plan to incorporate further static analysis techniques, such as infeasible path analysis, to improve *Scopda*'s treatment of trace sparsity in *improvement specifications*.

Chapter 6

Tracing Overhead and Observer Effects

Chapters 3, 4, and 5 developed a program analysis method, *DTRSO*, for tracing, analysing, and modifying a program to improve its concurrency performance. This chapter does not extend *DTRSO* further, instead we discuss a fundamental effect that dynamic program analysis has to account for, the observer effect inherent in program tracing and its implications.

6.1 Introduction

Execution tracing is used to capture precise and fine-grained data about a program's behaviour. These data can be used to better understand and optimise program behaviour and to enable automatic program analysis. It is especially useful for performance analysis, which depends on data from real executions.

However, execution tracing naturally has an *observer effect* (also known as a *probe effect*); the act of tracing itself perturbs the behaviour being observed. This is caused by overhead incurred by tracing. The effect can be substantial enough that the recorded behaviour is significantly distorted from the untraced behaviour. Any analysis, manual or automatic, on sufficiently distorted data is inherently flawed and may lead to invalid conclusions.

There are many dimensions along which overhead can be evaluated, we will consider three in particular:

Compute volume The amount of direct compute overhead (i.e. CPU cycles) introduced.

Memory volume The amount of memory consumed by the tracer. This indirectly incurs compute overhead by increasing page faults and can directly incur overhead if it consumes memory bandwidth in a bandwidth-constrained application.

Uniformity How uniformly overhead is incurred across threads (concurrent uniformity) and across time (temporal uniformity).

While all three inform the design of new execution tracers (such as which dimensions to prioritise for optimisation), they also elucidate the relative fragility of concurrent behaviour and analysis.

I argue that understanding the (non-)uniformity of overhead is essential to understanding its effect on program behaviour (Section 6.2). The common amortising mindset that "tracing adds x% overhead" is misguided as it does not account for the nuanced nature of overhead, instead assuming it is a consistent slow down of the program. Importantly, overhead can qualitatively affect a program's behaviour. This is not to suggest that calculating the amortised overhead is not valuable, but rather that it is only one of a set of measures of overhead that should be considered.

In this chapter we investigate the effect of tracing-overhead on *program concurrent behaviour* – how and when a program uses concurrency – and execution duration in real-world Java programs. We also consider how distorted-behaviour in traces affects the accuracy and validity of subsequent concurrency-performance analysis. Various *metrics* are suitable for exploring the effect of the overhead dimensions above. I focus on four key metrics:

MTU Mean thread usage – effectively the integral of thread use over time, divided by total time.

A-MTU Active threads MTU – *active* threads are those progressing computation at a given time.

W-MTU Waiting threads MTU – a thread is *waiting* when it is waiting on another thread to finish some work.

WCD Wall-clock duration of a program execution.

The key research questions addressed in this chapter are:

1. How does the observer effect inherent in execution tracing affect program concurrent behaviour?

2. How do the various dimensions of tracing-overhead (compute, memory, and uniformity) differ in their effects?

3. How is subsequent concurrency-performance analysis affected by the observer effect?

To investigate these questions, I augment the *Quilt* tracer, from Chapter 3, to generate configured additional overhead (Section 6.3). To investigate the effect of tracing-overhead on subsequent program analysis I use *Rehype* (Chapter 4, key points are summarised in Section 6.3.2), a concurrency-performance analyser.

Experimental results (Section 6.4) suggest that: the hypothesised observer effect does exist[1] and can significantly affect concurrent behaviour; overhead dimensions vary in their effects; and increased overhead reduces analysis accuracy. While the results are specific to concurrent behaviour and concurrency-performance analysis, I suggest that the observer effect is broadly relevant to many forms of dynamic analysis. Hypothetically, most forms of dynamic analysis, which rely on tracing, may be affected as, by definition, they analyse runtime behaviour which can change based on overhead. Practically, dynamic analyses

[1]Though the existence of different forms of the observer effect has been discussed in previous work (e.g. Gait [27]), see Section 6.5, the hypothesised effect I discuss here is the effect on concurrent behaviour. Whereas, previous works (and their hypothesises) have focused on other effects, such as the introduction of synchronization errors.

that directly work with concurrency or performance will be more significantly affected as a program's concurrency and performance behaviour will be most affected.

The aim of this chapter is to demonstrate the existence and relevance of the tracing observer effect on concurrent behaviour and analysis. I believe its relevance to dynamic analysis warrants further investigation. While the experimental evidence I present is a first step on this path of investigation, there is significant scope for further experimentation and analysis.

Finally, I position this chapter's work among related work in Section 6.5, discuss implications of the results, and implications of the experimental approach on the results, in Section 6.6, and conclude in Section 6.7.

Narrative context

In our narrative scenario (Section 1.2.2), Banjo uses *Quilt* to trace an execution of Paterson and *Rehype* to identify and quantify potential changes. A key aspect of both of these tools is that they provide insight into Paterson's resource usage, specifically thread usage, and not just wall-clock time. If Banjo only had superficial measurements, such as wall-clock duration, to base their decision on, they may not identify any significantly beneficial changes. Indeed, many of the beneficial changes *Rehype* identifies in Paterson may not significantly affect these superficial metrics. Yet those changes may be very beneficial in less obvious metrics, such as detailed thread usage, which can substantially affect performance, via metrics such as throughput, over time and at scale.

This chapter discusses the importance of more detailed metrics, especially as they relate to dynamic analysis systems.

6.2 Uniformity in Tracing

I wish to challenge an implied assumption in the statement "tracing adds x% overhead" (examples include [66][2], [57][3], and [76][4]), the assumption is that such overhead does not significantly affect program behaviour, it merely slows the program (i.e. wall/user-clock duration) – as though the processor, memory, and I/O speed had been slowed. Tracing-overhead is not equivalent to a slower processor, it is not uniform as the overhead is only incurred when events are traced and events are not traced uniformly. Thus, even given uniform per-event overhead, the impact of tracing-overhead will be non-uniform.

Furthermore, as tracing-overhead fundamentally interacts with the broader system, by consuming limited resources, it will have a greater impact at points of resource contention and bottlenecks. Any such impact is further amplified as points of contention have the greatest opportunity for distorting program behaviour.

Thus, while the apparent outcome of tracing may be x% longer execution duration, the qualitative effect on behaviour may be greater. For example, task-scheduling may be

[2] "Castor can processes [sic] 30M events per second if it owns a whole core, 1.6M / 30M is 5%. This translates to 5% application overhead, since that is what Castor steals from the application for logging."

[3] "Our evaluation also shows that TSVD introduces an acceptable overhead to the testing process — about 33% overhead for multi-threaded test cases while traditional techniques incur several times slowdowns."

[4] "Whereas sampling based tools can have up to 25% slowdown using 4kHz frequency, our tool `bcc-java` has a geometric mean of less than 5%."

affected[5], which could lead to deadlock via thread starvation[6]. Similarly, programs may change modes of operation, such as an adaptive bitrate streaming system [106] degrading quality of video segments due to increased overhead. So, though $x\%$ might be acceptable in some instances (e.g. for non-performance-analysis applications, such as record/replay), it may not be so innocuous in other instances.

6.3 Experimental Method

To investigate the effects of tracing overhead, I extend the tracer to generate configurable-overhead and use a concurrency-performance analyser to assess the effect of overhead on subsequent analysis.

6.3.1 Configurable overhead

For this chapter, I augment *Quilt* to generate configurable amounts of overhead so we can observe its effects – basically it is easier to add tracing-overhead than to remove it! I extend the trace-config with an *overhead-config* specifying that additional synthetic overhead is to be incurred when recording an event. The overhead-config can specify both additional *computation* (simulated by hash operations on a sequence of random bytes) and additional *memory allocation* (i.e. `malloc`[7]) by the tracing process, and also whether allocation has a corresponding `free` (allocating but not freeing memory enables isolating the effect of memory overhead from the allocation compute overhead). Ideally, we would like to be able to configure *uniformity of tracing-overhead*, however, as a uniform baseline is not possible (Section 6.2), I instead experiment with uniformity by adjusting which functions are instrumented (Section 6.4.1).

6.3.2 Concurrency-performance analyser

Chapter 4 described *Rehype*, a concurrency-performance analyser. In this chapter, we use *Rehype* to investigate the effect of tracing overhead on subsequent program analysis. As a refresher, I briefly summarise the relevant aspects of *Rehype* here.

Rehype takes a *trace-log*, generated by *Quilt*, and produces suggestions as to how the program's concurrency might be refactored to improve performance (such as inlining a task to reduce the amount of time threads in the program spend waiting). These suggestions are called *changes* and denoted using δ, here treated as maps from trace-logs to trace-logs.

Given a trace-log t, *Rehype* produces a set of changes Δ and an *estimated trace-log* $\delta(t)$, for each change $\delta \in \Delta$. These approximate the trace-log the program would generate with δ implemented on the source code. Then, given a performance metric μ and writing the value of μ for t as $\mu(t)$, *Rehype* calculates the estimated proportional improvement (EPI) of a change as:

$$\mathrm{EPI}^{\delta}_{\mu}(t) = (\mu(\delta(t)) - \mu(t)) \ / \ \mu(t) \tag{6.1}$$

[5]Some methods which make scheduling deterministic, such as Dthreads by Liu et al. [61], would be robust against such task-scheduling effects. However, concurrent determinism would not substantially affect the uniformity of overhead, given the fundamental nature of overhead uniformity discussed above.

[6]`https://wiki.sei.cmu.edu/confluence/display/java/TPS01-J.+Do+not+execute+interdependent+tasks+in+a+bounded+thread+pool`

[7]Recall that *Quilt* is programmed in Rust, not Java, and so allocates memory directly on the OS, not through the JVM.

The experimental situation below is slightly more complex as we have a range of traces t, obtained by varying *overhead-config* in the tracer, that similar changes δ are identified for.

6.4 Experimental results

I perform experiments to investigate the three research questions (see Section 6.1). I define the various overhead configurations used in experiments in Section 6.4.1, describe the experimental metrics in Section 6.4.2, report on the experiments in Section 6.4.3, and discuss experimental limitations in Section 6.4.4.

To investigate the real-world effects of tracing-overhead, I perform all experiments on an industrial Java API server, *Acme*. *Acme* uses a task-based concurrency model (centred on substantial usage of Java's standard thread pool interface, `ExecutorService`), has approximately 500k LoC, and has been in production for five years. Experiments execute a synthetic series-of-requests workload which simulates the standard use of *Acme* on a non-loaded machine. This provides a consistent workload[8] to evaluate the effects of various overhead configurations. As with other chapters, these experiments were run on a Linux benchmarking machine described in Section 2.6.

6.4.1 Configurations

I use two baseline configurations:

`base`: instruments only concurrency constructs (such as thread-pool and task events) necessary to capture concurrent behaviour.

`std`: the standard, but unoptimised, configuration used by *Rehype* to identify changes for *Acme* (this is the configuration used in Chapters 3 and 4). This configuration extends `base` by instrumenting all functions in *Acme* (i.e. the `com.acme` package, but not third-party library code). This is a useful reference configuration as it instruments functions necessary for *Rehype* to identify relevant changes, whereas `base` would trace an insufficient set of functions.

I denote additional overhead configurations using their overhead type and volume (Section 6.3.1):

`co-[x]`: Extends `std` with *compute* overhead by performing `[x]` hashes on each event; e.g. `co-2` performs two hashes each event.

`db-[x]`: Mirrors `co-[x]`, except it instruments third-party library functions that are primarily called during concurrent phases of *Acme* (specifically, database access functions) instead of those instrumented by `std`.

`ma-[x]` **and** `ml-[x]`: Extends `std` with *memory* overhead by allocating `[x]` bytes on each event; `ma` immediately frees the memory, while `ml` does not. These allow experiments to isolate memory use effects from the compute overhead of allocation (Section 6.3.1).

[8] *Acme* is not perfectly deterministic, for example, it generates random identifiers that rely on time-based seeds and it is a concurrent program in which task-scheduling may differ between runs. However, when executing the same workload multiple times, *Acme* produces consistently similar performance metrics (though, of course, not perfectly equal).

6.4.2 Metrics

I evaluate experiments on summary metrics of thread usage and wall-clock duration (WCD) as the overhead configuration is varied. I calculate the mean thread usage (MTU) across an execution for all occupied threads, *active* threads (A-MTU), and *waiting* threads (W-MTU). A thread is occupied when it is processing a task (as opposed to being idle in a thread pool); it is *active* when it is progressing computation; it is *waiting* when it is paused waiting for the result of another task. I derive additional metrics below to further evaluate the effects of overhead.

Metrics μ are naturally defined on trace-logs t. However, my experiments only vary configurations c and not other inputs, hence it is convenient to regard μ also as functions of c. The same applies to EPI (Section 6.3.2).

Relative effect metrics

I compare configurations c using the value of a metric relative to the `base` configuration (i.e. $\mu(c)$ / $\mu(\texttt{base})$). These derived metrics provide a direct measure of the distortion caused to concurrent behaviour by configurations' overhead. I define the relative effect (RE) of a metric μ as the absolute distance of the metric to the `base` version. That is, given a configuration c, the μ-relative effect (μ-RE) is:

$$\mu\text{-RE}(c) = \left| \frac{\mu(c)}{\mu(\texttt{base})} - 1 \right| \tag{6.2}$$

The ratio of MTU-RE to WCD-RE (shown in Fig. 6.2b) is useful as a measure of the proportional effect on concurrency and duration:

$$\text{MTU-RE-to-WCD-RE-ratio}(c) = \text{MTU-RE}(c) \text{ / } \text{WCD-RE}(c) \tag{6.3}$$

Measuring the effect on *Rehype* accuracy

To evaluate the effect of overhead on *Rehype*'s analysis accuracy, I compare the EPI (Eq. 6.1) for changes that are identified by `std` (as `base` does not record enough data) and additional overhead configurations. Specifically, for a given configuration, I calculate the relative EPI (REPI) for each change both it and `std` identify and then take the mean REPI (MREPI) across all such changes.

Assuming metric μ, I write Δ^c for the changes identified by *Rehype* in a trace-log under overhead configuration c, and define $\Delta = \Delta^c \cap \Delta^{\texttt{std}}$. The REPI of change $\delta \in \Delta$ indicates the reduction in accuracy for that change caused by the configuration's overhead:

$$\text{REPI}^{\delta}_{\mu}(c) = \text{EPI}^{\delta}_{\mu}(c) \text{ / } \text{EPI}^{\delta}_{\mu}(\texttt{std}) \tag{6.4}$$

The MREPI is then the average reduction in accuracy caused by the overhead over common changes Δ:

$$\text{MREPI}_{\mu}(c) = \frac{1}{|\Delta|} \sum_{\delta \in \Delta} \text{REPI}^{\delta}_{\mu}(c) \tag{6.5}$$

Concurrent profiles

Finally, the time-series metric of thread usage over a program execution (its *concurrent profile*) is useful to illustrate concurrent behaviour; its graphical form gives the shape of concurrency throughout an execution – i.e. when and how many threads are used. Two executions of the same program, with the same input and environment, should have almost identical concurrent profiles in the absence of overhead; Fig. 6.1b shows the concurrent profiles of three executions of *Acme*, under different overhead configurations.

6.4.3 Experiments

I devise separate experiments for each research question (see Introduction). The key takeaway is that: tracing-overhead can significantly distort program behaviour (to amplify or suppress concurrency), even when execution duration is largely unaffected.

RQ.1. How does the observer effect inherent in execution tracing affect program concurrent behaviour?

To investigate this question I use the plainest form of overhead, compute volume, and correlate it with a change in MTU, reflecting a distortion to the program's concurrency. This correlation is illustrated in Fig. 6.1a. The figure also shows that, for *Acme*, increased overhead decreases the MTU – making the program *appear* less concurrent. It is important to understand that while *Acme* does *behave* less concurrently (a greater proportion of the execution is spent on a sequential phase) given overhead, this is not reflective of its behaviour in a normal state without overhead (thus it merely *appears* less concurrent).

The intuition behind the reduction in apparent concurrency is that a particular sequential phase of execution within *Acme* generates more trace events and thus incurs more overhead. This is shown in Fig. 6.1b; the `co-1k` profile has an amplified sequential phase (approximately between 30% and 95% of the execution progress), while the `base` and `std` profiles appear more concurrent throughout. In those profiles, tracing incurs less overhead and so does not amplify the sequential phase as significantly.

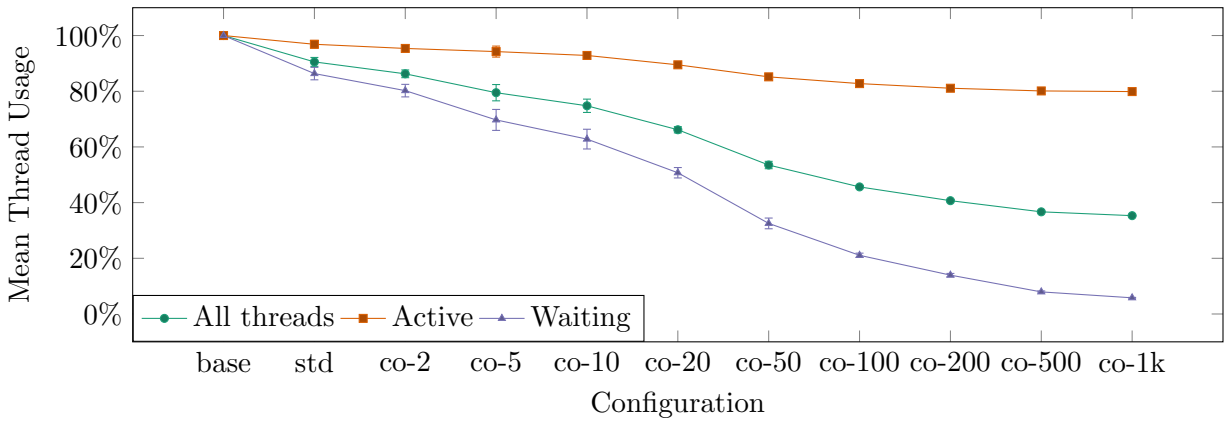
Fig. 6.1a also illustrates how concurrency distortions can affect the apparent concurrent efficiency of a program. The W-MTU decreases more than the A-MTU, suggesting (if taken at face value) that the program is more efficiently using concurrency (as *waiting* threads are an inefficient use of resources – see Chapter 4), than it does when executed without overhead.

Finally, Fig. 6.1c illustrates the change in maximum recorded concurrent thread use (MaxTU) due to overhead. A MaxTU change results from a task-scheduling change. This reflects a qualitative effect on program behaviour due to overhead. Note that the effect is not monotonic (though it appears so for the `db-*` configurations due to the increased overhead on background threads), rather, qualitative effects can be chaotic.

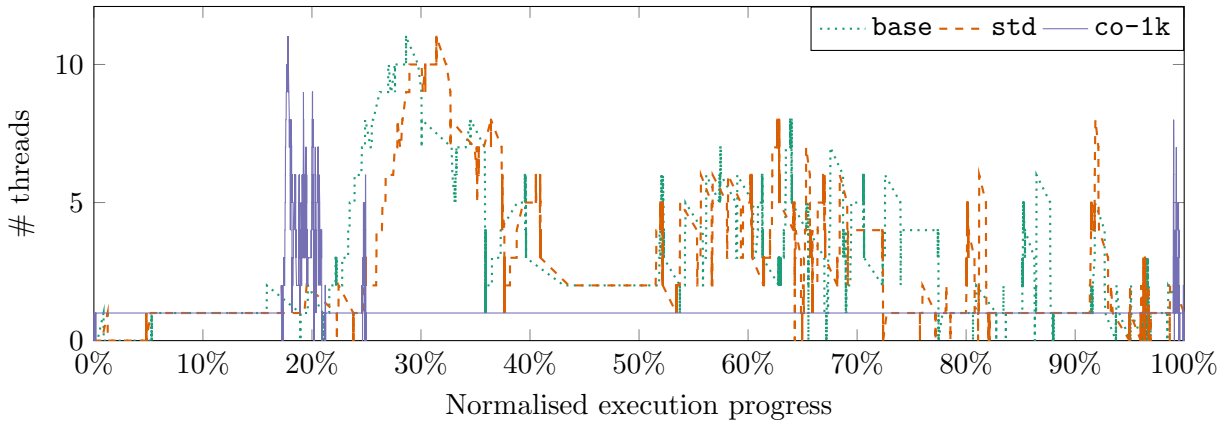
RQ.1. Answer: the observer effect can significantly distort the concurrent behaviour of a program. Furthermore, overhead can have qualitative effects that are not monotonic.

RQ.2. How do the various dimensions of tracing-overhead (compute, memory, and uniformity) differ in their effects?

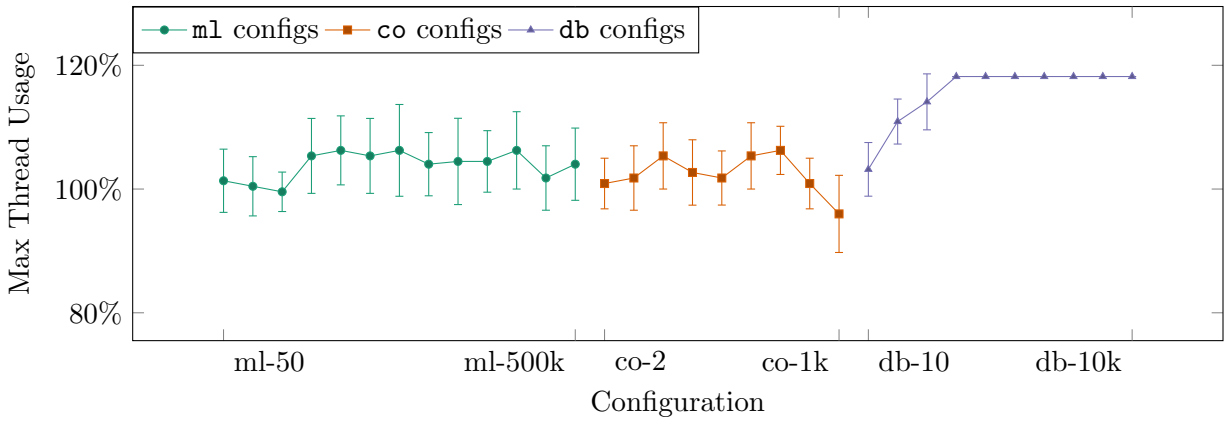
I discuss this question in two parts: (i) the subtleties between compute and memory overhead; and (ii) overhead uniformity and its fundamental nature. The difference between



(a) Relative Mean Thread Usage (MTU) for compute overhead configurations. Configurations approximately double overhead each increment.

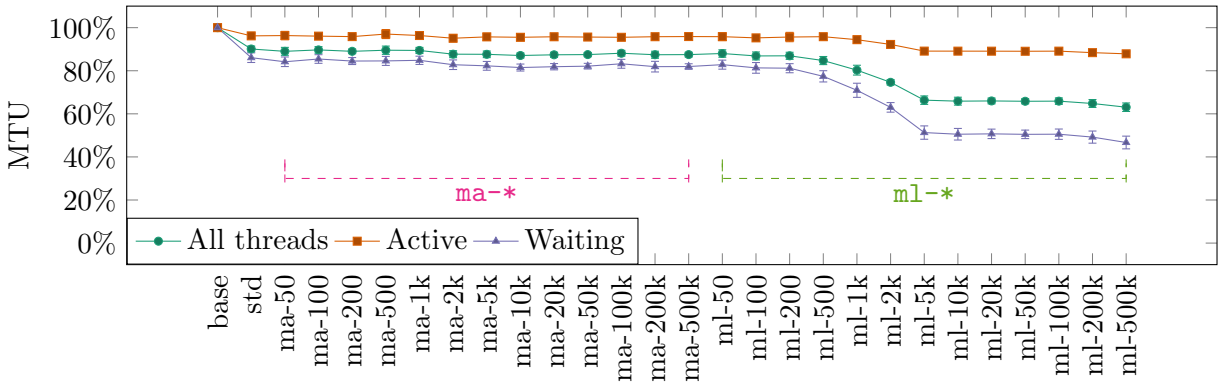


(b) Concurrent profiles (thread usage over normalised execution time) for the `base`, `std`, and `co-1k` configurations.

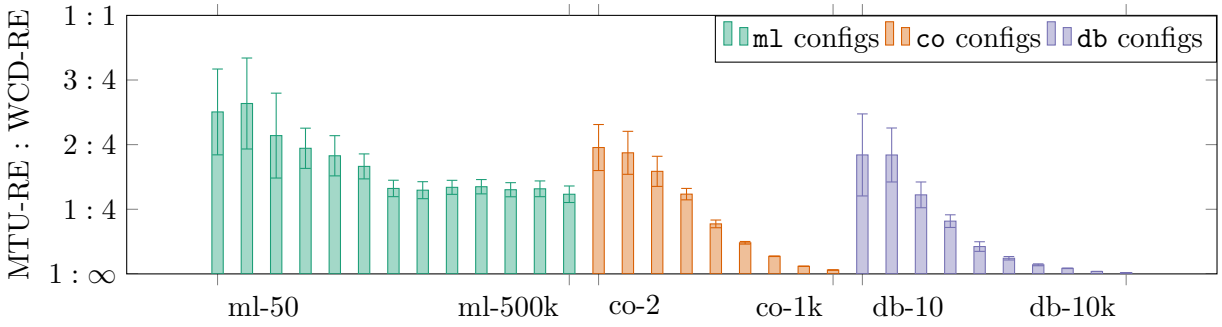


(c) Qualitative effect on program behaviour via the Maximum Thread Usage (MaxTU) relative to `base`. A different MaxTU reflects a different task-scheduling.

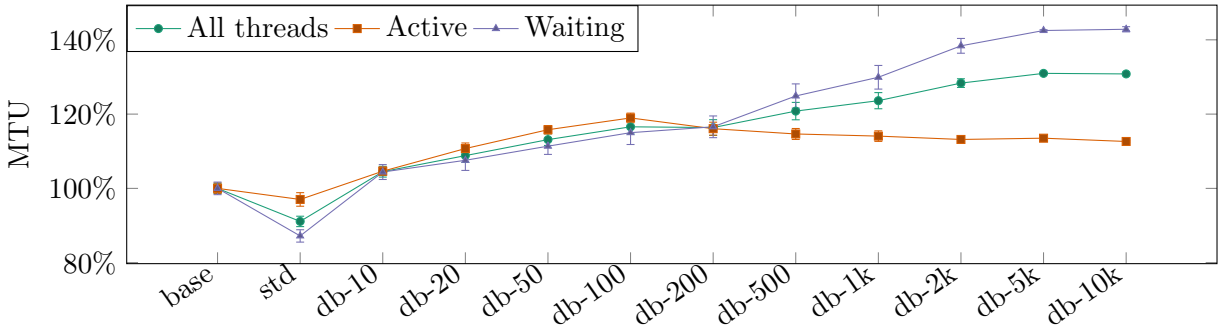
Figure 6.1: The effect of compute overhead on program concurrent behaviour. All configurations were executed 20 times.



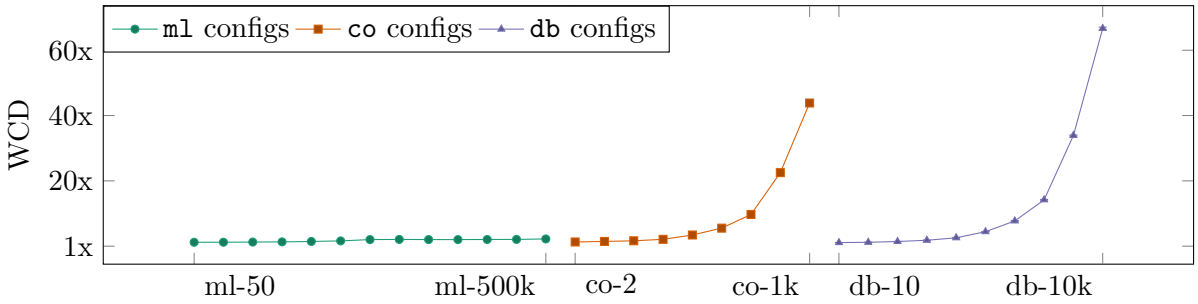
(a) Relative MTU for memory configurations.



(b) Ratio of MTU-RE to WCD-RE (Eq. 6.3).



(c) Relative MTU for configurations that amplify concurrency.



(d) Relative WCD for `ml-*`, `co-*`, and `db-*` configurations.

Figure 6.2: The effect of various overhead configurations. All configurations were executed 20 times.

compute and memory overhead is primarily exhibited by the difference between MTU and WCD. The effect on MTU is bounded, while the effect on WCD is not.

Given a maximum recorded concurrent thread use MaxTU > 1, the MTU is always less than it and greater than 1 (if the program does not deadlock), i.e. for a given configuration c, $1 <$ MTU(c) $<$ MaxTU. Moreover, the effect of increasing or decreasing overhead begins to level off as it nears these bounds. This effect is seen in Figs. 6.1a, 6.2a, and 6.2c.

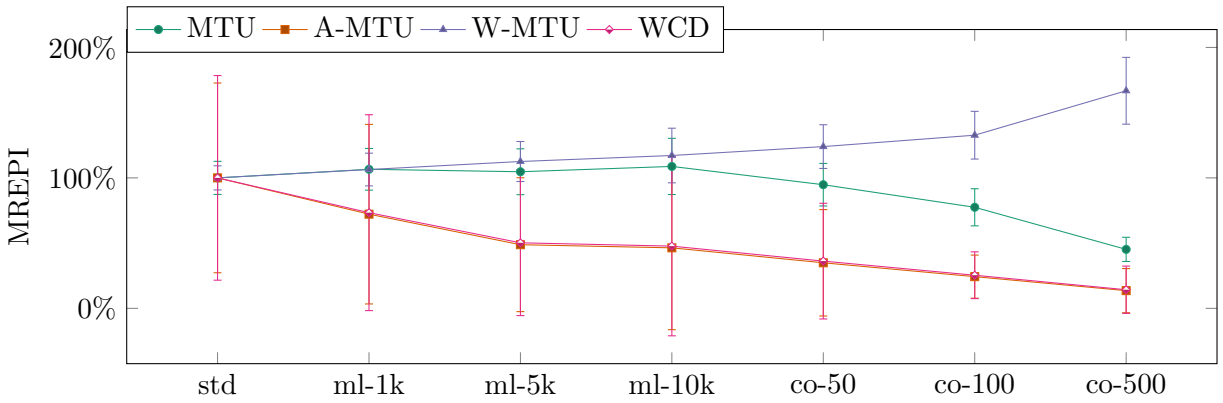
The WCD metric, however, has no such bound, it can continue to increase indefinitely (though it can never go negative). In fact, WCD-RE naturally accelerates as overhead increases and becomes the bottleneck in the program's execution, this is seen in Fig. 6.2d. The fact that `co-*` and `db-*` configurations affect duration significantly is unsurprising as they directly add to the compute time. What is interesting, however, is that `ml-*` configurations do not show the same significant increase in WCD, despite it having levels of MTU-RE equivalent to the `db-*` configurations. This difference is seen in the ratio of MTU-RE to WCD-RE (Fig. 6.2b).

I have argued that overhead **uniformity** cannot exist (Section 6.2). Yet, demonstrating the effect of uniformity in isolation would require a baseline with uniform overhead (i.e. uniform overhead incurred across time and across threads during the program execution by, for example, slowing all hardware components by some constant factor). A purely-uniform baseline cannot exist and a practically-uniform baseline cannot practically exist (practical uniformity could be constructed in a toy example, but not in a real-world system). As such, I must instead observe the effect of uniformity in other experiments, such as Fig. 6.1b. Here, the `std` and `co-1k` configurations track the same functions, however, the functions are not uniformly invoked across the execution. As `co-1k` incurs greater overhead, its profile shows that the functions are disproportionately invoked in the sequential phase.

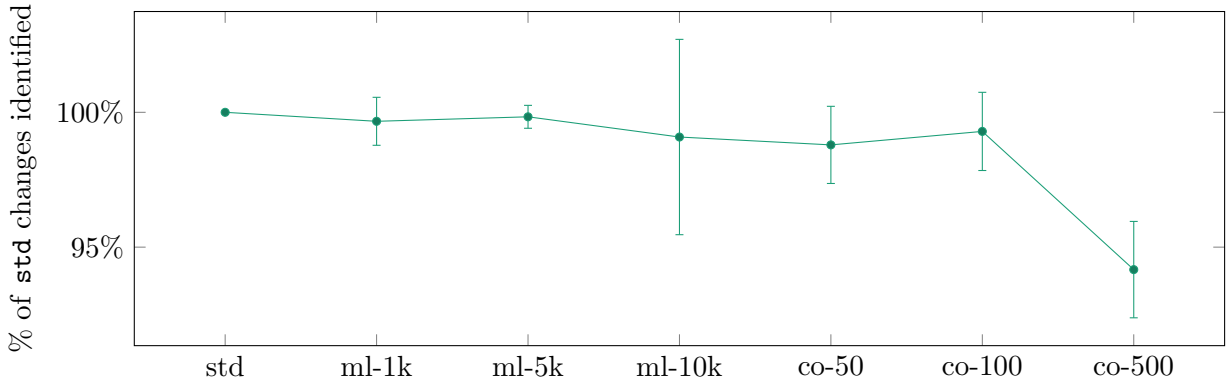
Uniformity, as a dimension of overhead, is important to consider as it intrinsically affects the behavioural distortion caused by overhead. It is present in all of these experiments as it determines where overhead is incurred. While the experiments so far have depressed concurrency (relative MTU less than 100%) the inverse distortion, an apparent increase in overall concurrency (relative MTU greater than 100%), can also be observed. For example, Fig. 6.2c shows the MTU for `db-*` configurations. These configurations increase the apparent concurrency by adding overhead to database access functions, which *Acme* primarily calls in background threads.

Whether distortion amplifies or depresses concurrency depends on which functions in the program generate the most trace events and when/where they are called (in concurrent or sequential phases). While this could, theoretically, result in uniformity if all threads produce identical events and overhead is perfectly uniform, such a scenario is unrealistic. In real programs, the number of events traced on each thread (concurrent uniformity) and at different times (temporal uniformity) will vary substantially causing overhead to be non-uniform (let alone naturally occurring non-uniformity due to tracing environment (e.g. cache behaviour)). Hence, in real programs, increased overhead correlates directly to increased concurrent behaviour distortion (i.e. increased MTU-RE).

RQ.2. Answer: increased compute and memory overhead correlates to increased MTU-RE and WCD-RE (though they are not proportional to each other), while uniformity controls the *shape* of the concurrent distortion. However, it is impractical to experimentally isolate overhead uniformity in real-world systems.



(a) The effect of overhead on *Rehype*'s estimation accuracy (Eq. 6.5). Note that A-MTU and WCD overlap almost entirely.



(b) The percentage of changes identified for the `std` configuration that are also identified for other configurations (i.e. $\left|\Delta^{c} \cap \Delta^{\texttt{std}}\right|$ / $\left|\Delta^{\texttt{std}}\right|$ for a given configuration c).

Figure 6.3: The effect of overhead on *Rehype*'s analysis. All configurations were executed 20 times.

RQ.3. How is subsequent concurrency-performance analysis affected by the observer effect?

Increased overhead reduces the accuracy of *Rehype*'s estimations (Section 6.3.2). Fig. 6.3 illustrates this via two charts. The first (Fig. 6.3a) illustrates the reduced accuracy of estimations as the MREPI (Eq. 6.5) of multiple performance metrics (MTU, A-MTU, W-MTU, and duration) for three `ml-*` and `co-*` configurations[9]. The second (Fig. 6.3b) shows the percentage of changes identified for the `std` configuration that were identified by other configurations. The change in the set of identified changes for configurations with additional overhead represents a qualitative effect on the analysis. This shows that overhead can have a qualitative effect on concurrent behaviour (task-scheduling Fig. 6.1c) and, separately, a qualitative effect on subsequent analysis.

The effect of overhead is compounded in estimated trace-logs (Section 6.3.2), as they

[9]The `db-*` configurations are omitted as they do not instrument enough functions for *Rehype* to identify changes.

inherit both the behaviour distortions from the trace-log they are based off and the reduced accuracy of estimation due to overhead. The end result is an estimate of program behaviour, for a given change, that is far from the real behaviour the program would exhibit with the change implemented and zero tracing overhead. This results in developers having reduced (human) confidence in the validity of the estimations which can lead to reduced adoption of changes (even if they are, in fact, beneficial).

That being said, I find that, generally, *Rehype* is able to identify many changes as beneficial (in binary terms) despite the concurrency distortions. I hypothesise that this is due to the relative-improvement approach of *Rehype*'s analysis.

RQ.3. Answer: analysis accuracy decreases as overhead increases, which matches intuition. This has been experimentally shown for *Rehype* and I suspect it holds for other analysis tools.

6.4.4 Limitations

These experimental results are limited to a single system *Acme*, running a single simulated workload, using four base metrics (MTU, A-MTU, W-MTU, and WCD), and analysed by *Rehype*. I argue that evaluation of tracer overhead to understand its various dimensions is important and that I have presented some initial results in these experiments. Future work should provide further rigour by evaluating more real-world systems, more program behaviour metrics, and more dynamic analysers.

6.5 Related work

The observer effect has been previously studied in relation to performance analysis by Mytkowicz et al. [72] and synchronisation errors by Gait [27]. Mytkowicz et al. analysed how it can perturb performance data and system measurements so that a manual analysis (by a "performance analyst") may draw incorrect conclusions. Gait investigated how it modifies the frequency of occurrence of synchronisation errors in concurrent programs (inserting variable length delays affects when (and whether) synchronisation errors occur). My work differs in that I specifically consider concurrent behaviour distortions and the effect on automatic concurrency-performance analysis. Indeed, in Gait's definition they state that "if the concurrent program being studied has no synchronization errors, then there is no probe effect."[10] (recall that probe effect is a synonym here for the observer effect), whereas in this work I consider the observer effect on concurrency, regardless of synchronisation errors.

Finally, the effect of various forms of overhead has been investigated in other fields. For example, the effect of cache overhead on allocation heavy programs (Grunwald et al. [33]) and sorting algorithms (LaMarca and Ladner [52]), the effect of network overhead on cluster computing (Martin et al. [65]), and the impact of address translation on performance (Zhou et al. [128]).

[10]This quote is from the second sentence of Gait's abstract [27]

6.6 Discussion

6.6.1 Functional effects of overhead

While in this chapter I have focused on distortions to program concurrent behaviour, the tracing observer effect may significantly affect functional aspects of certain programs. In general, tracing overhead could lead to deadlocks depending on the concurrency approach of a given program. For example, deadlock by thread-starvation – whereby all threads in a thread-pool are used and begin waiting for the result from another thread, with no threads left to process a necessary task – could be caused by increased overhead realigning thread scheduling (though such a deadlock would be indicative of poor practices that have enabled thread-starvation in this way).

Moreover protocols and systems that adjust their behaviour based on performance may be functionally affected by tracing-overhead. For example, adaptive video streaming [7] adjusts the quality of video segments based on the throughput of the client-server connection. While this is described in network terms in the literature, and in normal operation is driven by network conditions, if significant enough overhead (such as from tracing) were introduced in the client or server the apparent available throughput could be modified. This would result in a functional change to program operation – i.e. adjustment in video quality streamed and the associate effects within the program (such as time to process, etc). Such changes in functional operation could have a substantial effect on the efficacy of any program analysis performed on captured program behaviour data.

Investigating the potential functional effects of tracing overhead on different programs is an interesting line of inquiry. It would further elucidate the extent to which tracing can affect a target program.

6.6.2 Practical effects of overhead on developers

While some overhead effects (such as WCD) are directly observable by developers, others (such as MTU) that are equally important are not easily visible. The observability of effects is relevant to developers' usage of tracers, and confidence in subsequent analysis. For example, the MTU-RE to WCD-RE ratio (Fig. 6.2b) is relevant to developers' usage of tracers as they can directly observe WCD-RE (in terms of executions taking longer), but not MTU-RE. Many may reasonably assume that if execution duration is significantly affected, then other aspects of their programs will be similarly affected. Inversely, if the execution duration is not significantly affected, they may assume that other aspects are similarly unaffected. However, as shown in Section 6.4, this correlation does not always hold; distortions to concurrent behaviour and duration are not consistently proportional.

If analysis outputs are less accurate than developers expect, their confidence in the analyser (and potentially program analysis more broadly) will drop. Without confidence in program analysis, the many benefits to software development will not be realised. As such, reducing tracing overhead and mitigating its effects is essential to supporting adoption of dynamic analysis.

6.7 Conclusion

Execution tracers have an observer effect, the act of capturing data perturbs the data captured. Tracing overhead has numerous dimensions, I considered three: compute, memory, and uniformity. Experimental results show tracing-overhead distorts the concurrent behaviour of a program, quantitatively and qualitatively, and can affect the accuracy and validity of subsequent analysis. Fundamentally, uniformity cannot exist in tracing and non-uniformity must be accepted, understood, and mitigated in execution tracers. Furthermore, I have argued that uniformity is an essential, and understudied, dimension of overhead that warrants further investigation.

A starting mitigation for this observer effect in tracers is to highly optimise tracers, even if they are not being used in performance-sensitive environments. However, as overhead and non-uniformity is fundamental to tracing and it is therefore impossible to fully mitigate the effect, it is also important to measure and understand the kind and volume of distortions. For example, by measuring the amount of overhead introduced to various threads an analyser could determine the distribution of such overhead and potentially adjust for the overhead, or at a minimum alert the user to the potential impact of the overhead. Though the argument may become somewhat circular: measuring the overhead of tracing may itself introduce overhead.

Rehype is somewhat robust to the distortions introduced by tracing as its suggestions are based on *relative* improvement. However, this only provides a level of robustness in terms of *what* improvements it suggests, *Rehype*'s estimations are fragile to the effects of tracing overhead and non-uniformity. As noted in Chapter 4, the accuracy of estimations is important for practical use for such systems.

In future work I plan to evaluate tracing-overhead on more real-world programs and against more metrics. I also plan to investigate how tracing-overhead can qualitatively affect application functionality, such as in adaptive video streaming where overhead may cause video quality negotiation to have a different outcome.

Chapter 7

Discussion

Chapters 3 – 6 presented the core research contributions of this thesis. In this chapter, before concluding the thesis, we will discuss points of interest that either span multiple ideas across the previous chapters, or do not fit neatly within their bounds. Some of these ideas are specific to *DTRSO*, while others generalise beyond it to the field of dynamic analysis for concurrency optimisation and, more broadly, dynamic analysis for static optimisation. Similarly, while some ideas are concrete and clearly scoped, others are intentionally speculative. Finally, these discussions do not aim to form a particular narrative, rather, they are a set of discrete considerations some of which will, hopefully, spark ideas for other contexts and applications.

7.1 Runtime and language generalisation

Although this thesis has focused on Java, none of the algorithms, methods, or systems are Java specific. For example, *Rehype* is designed for generic programming constructs, such as tasks, and does not assume a specific implementation language or runtime engine. For *Rehype* to analyse applications based on other languages, only *Quilt* needs to be adjusted to trace those languages. This is analogous to adjusting the frontend of a compiler that produces an intermediate representation (IR); the trace-log is effectively *Rehype*'s IR. For example, for LLVM [54]-compiled languages I have implemented a proof-of-concept *Quilt* instrumenter (Chapter 3 discusses the instrumenter component of *Quilt*) as an LLVM pass[1]. Similarly, *Scopda* is designed so that, to support a new language, it only needs a new *language interface* to generate APGs from that language and render them back to the language.

In this way, both *Rehype* and *Scopda* have their own IRs (the trace-log and APG, respectively) which aim to represent general concepts of programs and programming languages, in dynamic and static contexts, respectively. Both IRs must, inherently, be flawed as they will naturally fail to represent concepts from other languages or runtime environments well, despite my best efforts to design them with generality in mind. I believe that generalised IRs that are designed for dynamic and static analysis are an interesting area for further research. While some may argue that existing compiler IRs (such as LLVM's IR) have achieved this for the static domain, I argue that a compiler IR has different aims to an analysis IR (though these distinctions may not always be obvious).

[1]This instrumenter is not discussed elsewhere in this thesis as it an early-stage proof-of-concept instrumenter and does not contribute further information to any of the primary contribution chapters.

For example, a compiler IR primarily aims to enable localised optimisations and final code generation. These goals necessitate constraints on the IR, such as ensuring full detail is retained for code generation. Whereas, *Scopda*'s form of analysis has different aims and therefore different constraints. *Scopda*'s IR (the APG) aims for ease of dynamic-to-static mapping and subsequent transformation for a specific type of optimisation at source-level. Given these differences in goals and constraints, it is natural that these different kinds of IR would warrant individual research efforts (though they will overlap at times).

In my implementation of *DTRSO*, I have taken a multi-phase approach that separates tracing, dynamic analysis, and dynamic-to-static mapping. The intention is that this will enable reuse of individual components for different languages, or even applications of the methods. However, despite this, some languages would pose further challenges for the fundamental model. For example, as *Scopda*'s approach assumes an APG representing a whole, fully resolved program, supporting dynamically typed (and dynamically loaded) languages, where types and functions can be mutated during runtime, would require adapting the fundamental design as they inherently cannot be statically fully resolved for analysis.

7.2 Concurrency model generalisation

Though this thesis has focused on task-based concurrency with thread-pool scheduling, the core methodological and theoretical contributions are generalisable beyond this context. The generalisation of *Quilt* to another concurrency model is reasonably straightforward, it would simply need to be extended to instrument events relevant to the concurrency model in question. *Rehype* would require more in-depth modification to accurately model a program's concurrent behaviour (e.g. by simulating a different task scheduling policy) and new change types would need to be created for each concurrency model, though, the core methodology of trace refactoring and estimation based on the trace-DAG should remain sound. *Scopda*'s only concurrency model specific component is the CTF (change transformation function) – CTFs would need to be implemented for each new change created for a concurrency model – otherwise it is independent of the concurrency model.

***Quilt*'s** tracing method is generally applicable; the core instrumentation and tracing framework is independent of the concurrency model. To support a new concurrency model, *Quilt* would need to be extended to instrument events specific to the concurrency model. For example, to support an async/await model, *Quilt* would need to instrument the async/await keywords or functions. Similarly, to support a coroutine based model, *Quilt* might need to include "continuation" events which indicate when a coroutine's continuation is executed on a different thread than it was suspended on. Supporting synchronisation by communication concurrency models (such as Go's channel-based concurrency [28]) presents an interesting challenge, on the surface it seems likely it would require *Quilt* to add data tags to messages sent, or to instrument the communication channel implementation, to enable resolving concurrency events across threads and across time in a program execution. However, this is not a topic I have investigated in depth and the reality may be quite different.

***Rehype*'s** core methods and theory are generalisable to a broader range of concurrency models, however, it would require non-trivial extensions and modifications to support

other concurrency models. For example, the trace–DAG encodes concurrency constraints using time-edges which are, in effect, "happens-before" edges. This constraint modelling for estimation is generalisable beyond task-based concurrency. Similarly, the theory and method underlying the transformation between trace-logs and trace-DAGs is applicable more broadly. Whereas, extensions would be required for other aspects of *Rehype* such as concurrency scheduling. For example, to accurately estimate programs built on a Fork-Join concurrency model, *Rehype* would need to simulate the Fork-Join task scheduling (i.e. work stealing), which it does not currently do. Finally, some of *Rehype*'s methods provide an informative basis, though likely not an implementation basis, for similar methods applied to other concurrency models. For example, task groups are a key aspect of *Rehype* that ensure suggested changes are statically viable. The core ideas behind task groups, such as creating sets of tasks based on the similarity of tasks' dynamic contexts, could be repurposed to other concurrency models, but are unlikely to be directly applicable beyond task-based concurrency models.

Extending *Rehype*'s concurrency model support is an interesting opportunity for future work. Such research could include supporting individual models (e.g. synchronisation by communication, such as Go's channel approach), support for integrating multiple concurrency models within one target program (e.g. a program that uses both task-based concurrency and synchronisation by communication concurrency), and improving the methods for estimating programs using specific models (e.g. improving the task-scheduling simulation for classical thread-pools or Fork-Join frameworks).

***Scopda*'s** key contributions centre on the dynamic to static translation, rather than on a specific concurrency model, and so is mostly independent of a target concurrency model. The primary work required to support a new concurrency model would be in implementing new CTFs for each optimisation identified for the concurrency model. Though no examples are immediately obvious, it is possible that some broader extension of *Scopda* might be necessary to implement CTFs for a new concurrency model.

Note that language variations of the task-based concurrency model, such as those utilising inbuilt `async/await` keywords, would require support via a *Scopda* language interface, as discussed in Section 7.1. However, this would be a language support extension, not a fundamental change in the concurrency model.

7.3 Simplicity and optimality; cost and benefit

Concurrency is hard, it always has been. New concurrent primitives, such as combinators, are ostensibly developed to make writing "good" concurrency easier. Practically this is not always achieved. Simply put, base primitives such as threads are conceptually straightforward but behaviourally sub-optimal, while higher-level constructs such as combinators are conceptually more difficult but behaviourally better. There is a trade-off between conceptual simplicity and behavioural outcome.

As many, if not most, industrial applications now use concurrency in some form, most developers will interact with concurrency. It is unrealistic to expect these developers to spend significant effort and time considering conceptually difficult concurrent constructs and approaches when "a simple thread will work well enough" (which may be true during initial development and only later become false).

Source-level fixes to these concurrency problems can be straightforward, but the cost, in developer time, of refactoring can seem unnecessary when the benefit is unclear. Hence the importance of accurately estimating the performance effects of changes to validate the cost-benefit trade-off of refactoring and generating automatic source-code patches where possible.

7.4 Estimation is good, so long as its accurate

A key approach I have developed, and advocated for, in this thesis is the use of estimation to identify improvements. This contrasts with merely identifying patterns that are *potentially* inefficient. I argue that estimation is a significantly more powerful approach (for performance analysis) as it can address patterns that are not always inefficient and can do it without significantly greater cost. Whereas, pattern matching analysers must either constrain themselves to patterns that are sure to be inefficient or perform some secondary step to confirm inefficiency (such as a controlled re-execution of a program).

However, for estimation to be effective, it must be very accurate. Decreased accuracy would likely have a quadratic[2] effect on confidence in results and thus practical effectiveness of the approach (without confidence, the estimations have no value).

Furthermore, estimation via dynamic-trace refactoring requires an *integrity model* – a model that programs adhere to so that the integrity of traces are maintained when being refactored (i.e. ensuring the refactoring of traces does not produce an impossible, or invalid, trace). Without an integrity model, invalid traces could be generated which would, in turn, invalidate the estimation. In *DTRSO* the integrity model is that the program will adhere to the task-based concurrency model (described in Chapter 2). Estimation methods that do not require such integrity models would be a significant research leap, but I do not currently see how they would be possible (without compromising integrity or accuracy).

7.5 Analysis at the developer's abstraction

DTRSO aims to generate developer-friendly (even developer-imitating) patches that are comprehensible and sensible to a developer. By operating at the developer's level of abstraction, at the source-code level, *DTRSO* can more naturally assist developers.

This approach is exhibited in each phase of *DTRSO*. *Quilt* traces function-level events, as opposed to lower-level instruction-based tracing or higher-level program event tracing (e.g. network-request events). *Rehype* performs trace refactorings that approximate simple changes to function invocations, as opposed to localised instruction re-orderings or large-scale architectural changes. Finally, a key aspect of *Scopda*'s IR (the APG) is that it maintains the source-code information so that it can generate patches that integrate into the existing source-code.

I argue that performing analysis at the developer's level of abstraction opens opportunities for various source-level optimisations, which can have greater individual effects than lower-level optimisations. In effect, there is significant power in being able to edit source-code, but this is only possible when operating at the developer's level of abstraction so that the developer will adopt suggested changes.

[2]This is an informed guess; I do not have a formal model for confidence.

7.6 Reversing the pipeline

There is significant potential in reversing the pipeline described in this thesis to instead estimate the effect of a source-code change created by a developer. This pipeline reversal could leverage many of the same methods (and implementations) as *DTRSO*, with appropriate adjustments as necessary.

Some code changes would be relatively trivial to estimate. For example, a change that removed a function call would likely be a relatively straightforward reversal of the pipeline. First, given a change and a trace, *Scopda* must identify which invocations of the function in the trace are affected by the change. Superficially this may appear easier than it is. In fact, it shares many of the challenges of the dynamic-to-static mapping *Scopda* currently performs. In particular, it must determine the appropriate invocations based on the static context of the call. Since the traces do not record specific call-sites, the same challenge as dynamic-to-static mapping, matching dynamic traces to static locations, exists. Once *Scopda* has determined the relevant invocations, *Rehype* can use its existing estimation method to determine the performance benefit of the change.

However, other changes would be significantly more challenging. Namely, inserting a new call to a function with non-trivial logic would require approximating what a trace of that function would contain (including event timings, concurrency operations, and so forth). This poses a interesting research challenge, likely requiring the use of statistical or machine learning methods to "sample" possible traces from the abstract space of all possible traces of the function. However, achieving high estimation accuracy would be significantly more difficult due to this extra layer of uncertainty and, as suggested above, decreased accuracy would likely have a significant effect on estimation confidence (and thus usage of any such tool).

7.7 The challenges of the real world

Performance analysis (of which dynamic analysis for concurrency optimisation is a subfield) is an interesting field that can improve software in a practical way. However, to have an impact, research performed in academia must be adopted by industry. Unfortunately, it appears that industry adoption is slower than the academic space might hope for. I believe this is because performance, in the general sense, is rarely a major pain point for many software companies. Regularly companies will prioritise new software features over improving the performance of existing features. This is further compounded by the fact that, in general, the software industry has been able to rely upon Moore's Law to compensate for poor programming practices.

Moreover, each piece of real-world software will evolve to contain its own varieties of programming edge cases as it tries to solve its own particular problem. It is thus worthwhile emphasising the importance of applicability to real-world software for academic research into program analysis, as contrived or benchmark examples may not include representative edge cases.

Finally, I want to reiterate my advocacy for source-level optimisations that use a human-in-the-middle (or developer-as-oracle) approach. To fully realise the benefit of program analysis research, it must be adopted by industry; one way to short-cut the challenges inherent in adoption is to create program analysis tools that empower developers to improve their software faster and more easily, rather than trying to provide guarantees

for all edge cases within a single program analyser. Source-level optimisations not only allow for more powerful high-level improvements, but also allow developers to directly work with program analysers. There are, of course, many other reasons program-analysis research achieves less-than-ideal adoption, but perhaps, in some cases, this approach could help.

Chapter 8

Conclusion

This thesis has presented the *theoretical development of a practical technique*, *DTRSO* (Dynamic-Trace-Refactoring for Static Optimisation), for automatically identifying performance improvements for concurrent programs. A key insight is that reducing concurrency can improve performance in certain, common, instances. Empirical results show that by removing specific instances of concurrency a large real-world, industrial API server's throughput can be more than doubled. Though a limited example, the existence of the inefficiencies in the real-world and the significant improvements achievable suggest that this is a real issue in modern concurrent software.

There are three key steps in *DTRSO*: tracing, analysis, and patch generation.

The key method of the tracer, *Quilt*, is a lockless buffer-exchange algorithm that enables tracing without blocking I/O calls on the target-program's worker threads. Maintaining low overhead, and minimal overhead on worker threads, is essential as tracers naturally have an observer effect – tracing a program affects its behaviour. This observer effect is non-trivial and, in many ways, non-obvious. Accounting for the effect of uniformity, a relatively opaque aspect of tracing overhead, on concurrent program behaviour, is important when developing tracers and analysers, and when considering the results from analysis.

Rehype, the concurrency-performance analysis component, refactors execution traces to approximate a trace the program would generate given some change. With this method, *Rehype* can use a single trace to accurately estimate the performance effect of multiple potential changes *without re-executing the program.*

The primary challenge for generating source-code patches is bridging the dynamic-to-static gap – translating *improvement specifications* which are based on dynamic data (execution traces) into source-code patches that rely on accurately calculated static (code) locations. To address this challenge, *Scopda* uses a new static code representation, an Abstract Program Graph (APG), which unifies the program's code structure, control-flow, local data-flow, and call-graph in a single graph. This unified approach is key to *Scopda*'s analysis *and* transformation approach.

I developed *DTRSO* to explore the field of Dynamic Analysis for Concurrency Optimisation and aspects of the broader topic of Dynamic Analysis for Static Optimisation – namely, generating static code patches from dynamic analysis. This exploration has demonstrated the potential that dynamic analysis for concurrency optimisation has for improving software and has also begun important work in tackling the key challenges of dynamic analysis for static optimisation. I maintain that concurrency optimisation represents an understudied portion of the broader field and that further research into both fields could open significant new avenues for improving software.

Findings

The thesis has achieved the following headline findings:

- It is possible to accurately estimate the effect of a source-level change to concurrency using a single execution trace.
- Generating source-code patches from dynamic analysis and dynamic data-based specifications is possible, but there are a number of practical challenges and theoretical limitations (such as disambiguation of call paths). While those discussed in this thesis are specific to *DTRSO*, I believe the theoretical considerations are likely similar for other possible approaches to the dynamic-to-static gap.
- Tracing has a natural observer effect that can significantly affect program behaviour. A key aspect of this effect derives from the uniformity of tracing overhead, something that has not, to the best of my knowledge, been discussed in depth before.

As well as these findings, numerous other theoretical and practical contributions to this area have been made and are discussed in their relevant chapters.

8.1 An exciting world

There are very exciting opportunities to leverage the power of dynamic analysis to improve modern software. To leave you with one example (also discussed in Chapter 7), it is theoretically possible to accurately estimate the performance effect of a source-level change made by a developer. One approach to this would involve, in essence, rearranging the method I have presented in this thesis. Given a set of representative trace-logs and a developer-written source-code change, translate the source-code change into a "dynamic change" to be made to each trace – effectively reversing *Scopda* – and then use *Rehype* to apply the "dynamic change" to the trace-logs and, thereby, estimate the effect of each change. This would make it possible to estimate the effects of a developer-written change under a variety of environmental conditions (e.g. "server under no load", "server under normal load", and "server under heavy load"), as each trace-log could represent a different set of environmental conditions. Of course, this is non-trivial and there are numerous challenges involved in such a project, but I believe they are solvable.

Excitingly, this would enable developers to *see the performance effects of their code in real-time as they write it.*

Bibliography

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[2] Marat Akhin and Mikhail et al. Belyaev. Kotlin language specification: Kotlin/Core. 2020.

[3] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479. URL `https://doi.org/10.1145/390013.808479`.

[4] Cyrille Artho. *Combining static and dynamic analysis to find multi-threading faults beyond data races*. PhD thesis, ETH Zurich, Konstanz, 2005. Diss., Technische Wissenschaften, Eidgenössische Technische Hochschule ETH Zürich, Nr. 16020, 2005.

[5] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, jun 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. URL `https://doi.org/10.1145/857076.857077`.

[6] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, 2010. doi: 10.1109/MC.2010.60.

[7] Abdelhak Bentaleb, Bayan Taani, Ali C. Begen, Christian Timmerer, and Roger Zimmermann. A survey on bitrate adaptation schemes for streaming media over http. *IEEE Communications Surveys & Tutorials*, 21(1):562–585, 2019. doi: 10.1109/COMST.2018.2862938.

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006. ISSN 0362-1340. doi: 10.1145/1167515.1167488. URL `https://doi.org/10.1145/1167515.1167488`.

[9] Luc Bläser. Practical detection of concurrency issues at coding time. In *the 27th ACM SIGSOFT International Symposium*, pages 221–231, New York, New York, USA, 2018. ACM Press.

[10] M. Braun, S. Buchwald, S. Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of static single assignment form. In *CC*, 2013.

[11] L. C. Briand, Y. Labiche, and J. Leduc. Tracing distributed systems executions using AspectJ. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 81–90, 2005.

[12] C++ coroutines. C++20 Coroutines. `https://en.cppreference.com/w/cpp/language/coroutines`, 2022.

[13] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting concurrency memory corruption vulnerabilities. In *the 2019 27th ACM Joint Meeting*, pages 706–717, New York, New York, USA, 2019. ACM Press.

[14] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[15] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40 (10):519–538, oct 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094852. URL `https://doi.org/10.1145/1103845.1094852`.

[16] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting Data Races Caused by Inconsistent Lock Protection in Device Drivers. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, March 2019.

[17] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 266–277. IEEE, April 2017.

[18] Fred Chow. Intermediate representation. *Communications of the ACM*, 56(12): 57–62, 2013.

[19] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970347. URL `https://doi.org/10.1145/2970276.2970347`.

[20] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[21] Clear Linux. Clear Linux* Project. `https://clearlinux.org/`, 2021. Accessed May 2021.

[22] Keith Cooper and Linda Torczon. *Engineering a Compiler: International Student Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 9780080472676.

[23] Dongdong Deng, Wei Zhang, and Shan Lu. Efficient concurrency-bug detection across inputs. In *the 2013 ACM SIGPLAN international conference*, pages 785–802, New York, New York, USA, 2013. ACM Press.

[24] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. *Kotlin Coroutines: Design and Implementation*, page 68–84. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450391108. URL `https://doi.org/10.1145/3486607.3486751`.

[25] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015. doi: 10.1109/ISPASS.2015.7095802.

[26] Matteo Frigo, Harald Prokop, Matteo Frigo, C Leiserson, Harald Prokop, Sridhar Ramachandran, Don Dailey, C Leiserson, I Lyubashevskiy, N Kushman, et al. The cilk project. *Algorithms*, page 8, 1998.

[27] Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986.

[28] golang. Go Language. `https://golang.org/`, 2021. Accessed September 2021.

[29] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[30] Brendan Gregg. Flame Graphs. `http://www.brendangregg.com/flamegraphs.html`, 2020.

[31] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Press, USA, 1st edition, 2011. ISBN 0132091518.

[32] Groovy. Apache Groovy Language. `https://groovy-lang.org/`, 2021. Accessed September 2021.

[33] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 177–186, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915984. doi: 10.1145/155090.155107. URL `https://doi.org/10.1145/155090.155107`.

[34] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

[35] Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute Menlo Park, CA, 1973.

[36] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `https://doi.org/10.1145/359576.359585`.

[37] Peter Hofer and Hanspeter Mössenböck. Efficient and accurate stack trace sampling in the java hotspot virtual machine. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, page 277–280, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327336. doi: 10.1145/2568088.2576759. URL https://doi.org/10.1145/2568088.2576759.

[38] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. Lightweight java profiling with partial safepoints and incremental stack tracing. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, page 75–86, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332484. doi: 10.1145/2668930.2688038. URL https://doi.org/10.1145/2668930.2688038.

[39] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck. Efficient tracing and versatile analysis of lock contention in java applications on the virtual machine level. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, page 263–274, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340809. doi: 10.1145/2851553.2851559. URL https://doi.org/10.1145/2851553.2851559.

[40] Intel PIN. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2021. Accessed june 2021.

[41] intellij. JetBrains IntelliJ IDEA. https://www.jetbrains.com/idea/, 2022. Accessed July 2022.

[42] Java Documentation. Java 8 `java.util.concurrent` documentation. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html, 2021. Accessed june 2021.

[43] Java Documentation CallerRuns. Java 8 documentation - ThreadPoolExecutor.CallerRunsPolicy. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.CallerRunsPolicy.html, 2021. Accessed june 2021.

[44] Java Flight Recorder. Java Flight Recorder. https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170, 2021.

[45] Java VisualVM. Java VisualVM. https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/profiler.html, 2020.

[46] javadev2019. JetBrains Develop Ecosystem Survey 2019, Java results. https://www.jetbrains.com/lp/devecosystem-2019/java/, 2019. Accessed July 2022.

[47] javadev2021. JetBrains Develop Ecosystem Survey 2021, Java results. https://www.jetbrains.com/lp/devecosystem-2021/java/, 2021. Accessed July 2022.

[48] Stephen C Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[49] Devin Kester, Martin Mwebesa, and Jeremy S Bradbury. How Good is Static Analysis at Finding Concurrency Bugs? In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 115–124. IEEE, June 2010.

[50] KProbes. Kprobe-based Event Tracing. `https://www.kernel.org/doc/html/latest/trace/kprobetrace.html`, 2021. Accessed june 2021.

[51] A. Krall. Efficient javavm just-in-time compilation. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, pages 205–212, 1998. doi: 10.1109/PACT.1998.727250.

[52] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1998.0985. URL `https://www.sciencedirect.com/science/article/pii/S0196677498909853`.

[53] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.*, pages 558–565, July 1978. URL `https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/`. 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). Also awarded an ACM SIGOPS Hall of Fame Award in 2007.

[54] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

[55] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, page 36–43, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132883. doi: 10.1145/337449.337465. URL `https://doi.org/10.1145/337449.337465`.

[56] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. Efficient memory traces with full pointer information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341356. doi: 10.1145/2972206.2972220. URL `https://doi.org/10.1145/2972206.2972220`.

[57] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection. In *the 27th ACM Symposium*, pages 162–180, New York, New York, USA, 2019. ACM Press.

[58] Yanze Li, Bozhen Liu, and Jeff Huang. SWORD: A Scalable Whole Program Race Detector for Java. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 75–78. IEEE, May 2019.

[59] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390590X.

[60] Linux perf. `perf`: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2021.

[61] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 327–336, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043587. URL `https://doi.org/10.1145/2043556.2043587`.

[62] lldb. The LLDB Debugger. `https://lldb.llvm.org/`, 2022. Accessed July 2022.

[63] llvmpgo. LLVM-Clang User Manual Profile-Guided Optimization. `https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization`, 2022. Accessed July 2022.

[64] Derrick Lockwood, Benjamin Holland, and Suresh Kothari. Mockingbird: a framework for enabling targeted dynamic analysis of java programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 39–42. IEEE, 2019.

[65] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *SIGARCH Comput. Archit. News*, 25(2):85–97, May 1997. ISSN 0163-5964. doi: 10.1145/384286.264146. URL `https://doi.org/10.1145/384286.264146`.

[66] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. *SIGPLAN Not.*, 52(4):693–708, April 2017. ISSN 0362-1340. doi: 10.1145/3093336.3037751. URL `https://doi.org/10.1145/3093336.3037751`.

[67] Chris McDonald and Matthew Heinsen Egan. Communicating using program traces. In *SPLICE Spring 2019 Workshop, CS Education Infrastructure for All II: En-abling the Change*, 2019.

[68] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[69] Michal Zalewski. American Fuzzy Lop (AFL). `https://lcamtuf.coredump.cx/afl/`, 2022. Accessed July 2022.

[70] Jerome Miecznikowski and Laurie Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *International Conference on Compiler Construction*, pages 111–127. Springer, 2002.

[71] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[72] Todd Mytkowicz, Peter F Sweeney, Matthias Hauswirth, and Amer Diwan. Observer effect and measurement bias in performance analysis. *Computer Science Technical Reports CU-CS-1042-08, University of Colorado, Boulder*, 2008.

[73] Nicholas Nethercote and Alan Mycroft. Redux: A Dynamic Dataflow Tracer. *Electronic Notes in Theoretical Computer Science*, 89(2):149–170, October 2003.

[74] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007.

[75] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Principles of program analysis. In *Springer Berlin Heidelberg*, 1999.

[76] Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján. Profiling and tracing support for java applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 119–126, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362399. doi: 10.1145/3297663.3309677. URL `https://doi.org/10.1145/3297663.3309677`.

[77] A Nistor, Qingzhou Luo, M Pradel, T R Gross, and D Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the 38th International Conference on software engineering*, pages 727–737. IEEE, 2012.

[78] S. Oaks. *Java Performance: In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond*. O'Reilly Media, 2020. ISBN 9781492056089. URL `https://books.google.co.uk/books?id=Q3_QDwAAQBAJ`.

[79] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. 2004.

[80] Indigo Orton and Alan Mycroft. Tracing and its observer effect on concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2021, page 88–96, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386753. doi: 10.1145/3475738.3480940. URL `https://doi.org/10.1145/3475738.3480940`.

[81] Indigo Orton and Alan Mycroft. Refactoring traces to identify concurrency improvements. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, page 16–23, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385435. doi: 10.1145/3464971.3468420. URL `https://doi.org/10.1145/3464971.3468420`.

[82] Indigo Orton and Alan Mycroft. Source code patches from dynamic analysis. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, page 1–8, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385435. doi: 10.1145/3464971.3468416. URL `https://doi.org/10.1145/3464971.3468416`.

[83] Dileep Kumar Pattipati, Rupesh Nasre, and Sreenivasa Kumar Puligundla. Opal: An extensible framework for ontology-based program analysis. *Software: Practice and Experience*, 2020.

[84] Mathias Payer, Enrico Kravina, and Thomas R. Gross. Lightweight memory tracing. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 115–126, San Jose, CA, jun 2013. USENIX Association. ISBN 978-1-931971-01-0. URL `https://www.usenix.org/conference/atc13/technical-sessions/presentation/payer`.

[85] PCGaming. PCGaming Wiki – Fallout 3 multicore crash and fix. `https://www.pcgamingwiki.com/wiki/Fallout_3#Game_crashes_randomly`, 2008. Accessed July 2022.

[86] PCGamingF3. PCGaming Wiki – Fallout 3. `https://www.pcgamingwiki.com/wiki/Fallout_3`, 2008. Accessed July 2022.

[87] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 16–27, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 0897913647. doi: 10.1145/93542.93550. URL `https://doi.org/10.1145/93542.93550`.

[88] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[89] Ernest Pobee and W K Chan. AggrePlay: efficient record and replay of multi-threaded programs. In *the 2019 27th ACM Joint Meeting*, pages 567–577, New York, New York, USA, 2019. ACM Press.

[90] Ernest Pobee, Xiupei Mei, and W K Chan. Efficient Transaction-Based Deterministic Replay for Multi-threaded Programs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 760–771. IEEE, 2019.

[91] Michael Pradel and Thomas R Gross. Fully automatic and precise detection of thread safety violations. *ACM SIGPLAN Notices*, 47(6):521–530, 2012.

[92] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, volume 1999, pages 81–91, 1999.

[93] Christoph Rathfelder, Benjamin Klatt, Kai Sachs, and Samuel Kounev. Modeling event-based communication in component-based software architectures for performance predictions. *Software & Systems Modeling*, 13(4):1291–1317, 2014.

[94] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[95] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin. An efficient and scalable platform for java source code analysis using overlaid graph representations. *IEEE Access*, 8: 72239–72260, 2020.

[96] Andrea Rosà, Eduardo Rosales, and Walter Binder. Analysis and optimization of task granularity on the java virtual machine. *ACM Trans. Program. Lang. Syst.*, 41 (3), jul 2019. ISSN 0164-0925. doi: 10.1145/3338497. URL https://doi.org/10.1145/3338497.

[97] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi: 10.1145/73560.73562. URL https://doi.org/10.1145/73560.73562.

[98] Ruby Tracer. Ruby/Tracer. https://github.com/ruby/tracer, 2021. Last accessed June 2021.

[99] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *LISP AND SYMBOLIC COMPUTATION*, pages 288–298, 1993.

[100] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. *ACM SIGPLAN Notices*, 49(10):473–489, 2014.

[101] Malavika Samak and Murali Krishna Ramanathan. Synthesizing tests for detecting atomicity violations. In *Proceedings of the 2015 10th Joint Meeting on foundations of software engineering*, pages 131–142. ACM, 2015.

[102] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. *ACM SIGPLAN Notices*, 50(6):175–185, 2015.

[103] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische annalen*, 92(3):305–316, 1924.

[104] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):540–582, 2000.

[105] Sukhdeep Sodhi, Jaspal Subhlok, and Qiang Xu. Performance prediction with skeletons. *Cluster Computing*, 11(2):151–165, 2008.

[106] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28 (4):1698–1711, 2020. doi: 10.1109/TNET.2020.2996964.

[107] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. Efficient coroutines for the java platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, page 20–28, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302692. doi: 10.1145/1852761.1852765. URL https://doi.org/10.1145/1852761.1852765.

[108] James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3), July 2013. ISSN 0360-0300. doi: 10.1145/2480741.2480743. URL https://doi.org/10.1145/2480741.2480743.

[109] Sebastian Steenbuck and Gordon Fraser. Generating Unit Tests for Concurrent Classes. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 144–153. IEEE, 2013.

[110] J Subhlok and Qiang Xu. Automatic construction of coordinated performance skeletons. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–5. IEEE, 2008.

[111] Herb Sutter. The free lunch is over a fundamental turn toward concurrency in software. 2005.

[112] Alexander Tarvo and Steven Reiss. Automated analysis of multithreaded programs for performance modeling. In *Proceedings of the 29th ACM/IEEE international conference on automated software engineering*, pages 7–18. ACM, 2014.

[113] Valerio Terragni and Shing-Chi Cheung. Coverage-driven test code generation for concurrent classes. In *Proceedings of the 38th International Conference on software engineering*, pages 1121–1132. ACM, 2016.

[114] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017. doi: 10.1109/MCSE.2017.29.

[115] Frank Tip. Infeasible paths in object-oriented programs. *Science of Computer Programming*, 97:91–97, 2015.

[116] Peter Trapp. *Performance Improvements Using Dynamic Performance Stubs*. PhD thesis, De Montfort University, 2011.

[117] Peter Trapp and Christian Facchi. How to Handle CPU Bound Systems: A Spezialization of Dynamic Performance Stubs to CPU Stubs. *undefined*, 2008.

[118] Peter Trapp and Christian Facchi. Main Memory Stubs to Simulate Heap and Stack Memory Behavior. In *CMG*, 2010.

[119] Peter Trapp, Markus Meyer, and Christian Facchi. Using CPU Stubs to Optimize Parallel Processing Tasks: An Application of Dynamic Performance Stubs. In *2010 Fifth International Conference on Software Engineering Advances (ICSEA)*, pages 471–476. IEEE, 2010.

[120] Peter Trapp, Markus Meyer, Christian Facchi, Helge Janicke, and Francois Siewe. Building CPU stubs to optimize CPU bound systems: An application of dynamic performance stubs. *International Journal on Advances in Software*, 2011.

[121] R G Urma, M Fusco, and A Mycroft. Modern Java in Action: Lambdas, streams, functional and reactive programming, 2018.

[122] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[123] Andreas Wilhelm, Bharatkumar Sharmay, Ranajoy Malakary, Tobias Schule, and Michael Gerndt. Symbolic analysis of assembly traces: Lessons learned and perspectives. In *2015 IEEE 6th International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 7–12. IEEE, February 2015.

[124] Paweł T Wojciechowski. Concurrency Combinators for Declarative Synchronization. In *Programming Languages and Systems*, pages 163–178. Springer, Berlin, Heidelberg, Berlin, Heidelberg, November 2004.

[125] Xing Wu. *Scalable Communication Tracing for Performance Analysis of Parallel Applications*. PhD thesis, ProQuest Dissertations Publishing, North Carolina State University, 2013.

[126] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: Event-based tracing to measure mobile application and platform performance. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013.

[127] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. Conseq: Detecting concurrency bugs through sequential errors. *SIGARCH Comput. Archit. News*, 39(1):251–264, March 2011. ISSN 0163-5964. doi: 10.1145/1961295.1950395. URL `https://doi.org/10.1145/1961295.1950395`.

[128] Yufeng Zhou, Xiaowan Dong, Alan L. Cox, and Sandhya Dwarkadas. On the impact of instruction address translation overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 106–116, 2019. doi: 10.1109/ISPASS.2019.00018.