Finding the needle in a haystack: chasing rarely occurring bugs in concurrent software

Faheem Ullah

DISS. ETH No. 23552

Finding the needle in a haystack: chasing rarely occurring bugs in concurrent software

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

FAHEEM ULLAH

M.Sc., Software Engineering, University of York
born on January 01, 1982,
citizen of Pakistan

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Prof. Dr. Sally A. McKee, co-examiner
Prof. Dr. Roland Wismüller, co-examiner

2016

To my wife and loving parents

Abstract

Ensuring software correctness is a challenging task to accomplish with the increasingly large and complex functional as well as non-functional requirements of modern software systems. Achieving correctness is becoming especially challenging as concurrent and multi-threaded programs are becoming more prevalent with the widespread adoption of multi-core machines. Anomalies associated with concurrent software are notoriously hard to detect and debug because they manifest in low probability interleavings of the program threads. Concurrent anomalies surface in executions that do not correspond to a sequential execution of the program statements. In addition to the same problems that exist when analysing sequential programs, software development tools for parallel systems must handle the large number of execution engines (cores) that result in different (possibly non-deterministic) schedules for different executions of the program.

The goal of this dissertation is to provide a foundation for the debugging of parallel software to uncover *rare anomalies* that appear in program executions. The techniques presented as part of this dissertation are based on analysing the dynamic behaviour of software systems. Although dynamic analysis is limited to analysing a single execution instance on a given input, however, it offers diagnostic precision. This dissertation presents techniques for detecting *functional* as well as *non-functional* anomalies in concurrent software. The techniques are able to *identify* and *localise* anomalies to a small part of the program code that may be inspected by programmers. The technique for detecting functional concurrency bugs employs dynamic control flow information to detect and consequently localise anomalous behaviour to a small part of the program code, while requiring a single failing execution of the program. The technique for detecting non-functional anomalies is able to detect performance issues in large scale software by utilising the amount of time consumed by different parts of the program. Experimental evaluation of both the functional and non-functional concurrency bug detection techniques show that the techniques are effective in detecting anomalies in well known large-scale software systems.

This dissertation contributes towards a better understanding of the manifestation conditions for real-world concurrency bugs.

Zusammenfassung

Die Erstellung korrekter Software ist eine herausfordernde Aufgabe vor allem in Anbetracht der zunehmenden Grösse und Komplexität der funktionalen und nicht-funktionalen Anforderungen an moderne Softwaresysteme. Diese Korrektheit wird immer schwieriger zu erreichen, da durch die grossflächige Verbreitung von Multicore-Prozessoren auch die Verbreitung von Programmen mit nebenläufigen oder parallelen Abläufen ebenfalls rasant zunimmt. Anomalien, die im Zusammenhang mit nebenläufiger Programmausführung stehen, sind sehr schwer zu entdecken und einzugrenzen, da sie oft nur unter bestimmten Verschachtelungen der nebenläufigen Programmabläufe auftreten und das Auftreten dieser Verschachtelungen meist eine tiefe Wahrscheinlichkeit hat. Diese nebenläufigen Anomalien treten in solchen Programmausführungen auf, welche nicht einer sequentiellen Programmausführung entsprechen. Entwicklungswerkzeuge müssen nun nebst zu den bereits existierenden Funktionen zum Analysieren von sequentiellen Programmen, auch die unterschiedlichen, möglicherweise nichtdeterministischen Ausführungsreihenfolgen der nebenläufigen Programme beherrschen.

Das Ziel dieser Dissertation ist eine Grundlage für die Fehlersuche in nebenläufigen Programmen zu schaffen, welche ermöglicht selten auftretende Anomalien in Programmabläufen aufzudecken. Die Techniken, welche als Teil dieser Dissertation präsentiert werden, basieren auf der Analyse des dynamischen Verhaltens von Softwaresystemen. Eine dynamische Analyse, obschon limitiert auf die Analyse einer einzigen Ausführung per Eingabe, bietet eine diagnostische Präzision. Diese Dissertation präsentiert Techniken für die Erkennung von funktionalen und nicht-funktionalen Anomalien in nebenläufigen Programmen. Diese Techniken sind fähig Anomalien zu identifizieren und diese auf einen kleinen Teil des Quellcodes genau zu einzugrenzen, so dass dieser dann durch den Programmierer untersucht werden kann. Die Technik um funktionale Nebenläufigkeitsfehler zu erkennen verwendet dynamische Kontrollflussinformationen. Aufgrund dieser kann das fehlerhafte Verhalten auf einen kleinen Teil des Quellecodes genau eingegrenzt werden. Dazu wird eine fehlerhafte Ausführung des Programms benötigt. Die Technik um nicht-funktionale Anomalien zu erkennen ist fähig Performanceprobleme in grossen Softwaresystemen zu erkennen, indem der Zeitverbrauch der unterschiedlichen Programmteile gemessen und ausgewertet wird. Die experimentelle Auswertung dieser beiden Techniken zeigt deren Effektivität im Erkennen von Anomalien anhand grossen, bekannten Softwaresystemen.

Diese Dissertation leistet einen Beitrag zum besseren Verständnis der Bedingungen unter denen sich Nebenläufigkeitsfehler in Software manifestieren.

Acknowledgments

I am greatly indebted to my advisor Prof. Dr. Thomas Gross for selecting me and providing me with an opportunity to improve my qualification. I am most thankful to him for placing his trust in me and for his patience, wise guidance, and timely encouragement. Without his constant support and encouragement I would not be where I stand today. I am also grateful to Prof. Dr. Karsten Schwan (late) of Georgia Institute of Technology for his feedback, and for graciously agreeing to be on my thesis committee. I am thankful to my co-examiners Prof. Dr. Sally McKee and Prof. Dr. Roland Wismüller for their valuable feedback on my thesis. In addition, I am extremely grateful to my masters thesis supervisor Dr. Manuel Oriol, who made it possible for me to come to ETH Zurich.

I thank my siblings who helped me in various ways and gave me assistance beyond measure. Without their help and support I would not be who I am. I owe it all to my eldest brother Tasleem Ullah who encouraged me to pursue higher education. I thank my siblings for their words of encouragement, and my parents for their constant prayers and belief in me.

I am thankful to my wife for her unwavering faith in me, and for her constant words of encouragement during my frequent moments of doubt. I owe her more than I can ever express. To our daughter: I can not wait to meet you.

I thank my office mate Luca Della Toffola who always urged me to keep courage on occasions that were crucial in my academic life. I am highly indebted to my past colleagues Nicholas Matsakis and Christoph Angerer for their advice. I will always cherish the highly addictive evenings — journeying through the mystical lands of dungeons and dragons in your company. I am especially indebted to Christoph Angerer for his insightful discussions - without his ideas and advice, my thesis would not exist. I would like to extend special thanks to my colleagues Zoltán Majó and Martin Bättig for being ever helpful. A big thank you to the rest of my colleagues and office mates (past and present) Animesh Trivedi, Antonio Barresi, Remi Meier, Michael Fäs, Aristeidis Mastoras, Ivana Unković, Oliver Trachsel, Stephanie Balzer, Mathias Payer, Albert Noll, Michael Pradel, Daniele Spampinato, Victoria Caparros, and Georg Ofenbeck. Thank you for your company, you made it possible for me to persevere all these years.

Lastly, I would also like to extend my thanks, to the many researchers whose works have undoubtedly helped shape my views.

Contents

1 Introduction

The need for better engineered and correct software is ever greater as software systems have become ubiquitous. Software controls a wide array of systems, including medical, aviation, industrial, and running inside the controls of more personal and safety-critical systems, such as the cars we drive, or the airplanes and trains we take for journeys. Programming errors that lead to software failure in such systems could have detrimental consequences.

The term “software engineering” was coined almost five decades ago when the software industry set out to search for a silver bullet to slay the monsters of missed deadlines, blown budgets and flawed products [97]. Nearly fifty years later, this search is still on, and software troubles continue to haunt us. The software community has been actively involved in researching new testing methodologies to counter the problems, however, errors still manage to slip into published software.

Software testing and debugging is an indispensable part of software development. It is a labour-intensive and financially expensive activity that consumes up to 50 percent of the elapsed time and more than 50 percent of the total software development costs [87]. According to a recent report from the National Institute of Standards and Technology, software bugs are so prevalent, and at the same time so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product of the country [92]. Software bugs can remain undetected in deployed systems for a long time as Knuth mentions in his volume on *Sorting and Searching* [64]. He points out that even though the first binary search was published in 1946, it took fourteen more years before the first version of binary search that worked correctly for all N was published. The story of binary search does not end there. More than four decades later, towards the end of 2004, it was discovered that the implementation of binary search in the Java Development Kit contained a bug [128]. The algorithm for the implementation was proven and tested to be correct in widely used and accredited books on how to write programs correctly [12]. The bug comes from computing the midpoint for binary search when deciding on whether to search the left or right sub-half of the list for a given search item. The incorrect statement to calculate the mid point in a list as it appeared in the binary search implementation for the Java Development Kit is as follows:

```
...
int mid = (low + high) / 2;
...
```

Calculating the midpoint (variable *mid* in the code snippet) in such a manner may cause an integer overflow if the sum of the two indices is larger than a value that can fit in a 32 bit integer. In the case when the addition produces a number larger than 32 bits and causes an overflow, the resulting value for midpoint is erroneously calculated as a negative number. The program then attempts to access a non-existent index in the list and causes the program to fail. The bug manifests itself for lists with 2^{30} or more elements. Lists with such a large number of elements were perhaps inconceivable a few decades ago when the algorithm for binary search was written and tested, however, with the growing requirements for data processing and computational needs, such lists are commonplace.

Software correctness has become more difficult to realise with the vast array of programming languages, hardware, and software platforms that have proliferated over the years. Specifically with the reality of multicore hardware, even ordinary desktop machines are capable of tackling computation-intensive problems that would previously require specialised high-performance mainframes. However, the advancement in technology comes at a price. The advent of multi-core processors has often been blamed for the degradation in reliability of software systems [38]. The rapid growth in multi-core platforms pronounces the ever greater need for programmers to write concurrent and multi-threaded programs to utilise the available resources more efficiently. Writing multi-threaded program is hard. As a consequence, concurrent software systems are known to be error prone, because it is difficult to reason about the behaviour and correctness of such systems [16]. Programmers tend to think sequentially and are prone to making mistakes when writing concurrent applications [76]. Further, the non-determinism that is inherent in concurrent programs renders the task of writing correct programs extremely challenging. Concurrent programs often exhibit wrong behaviour due to unintended interference among the multiple threads of a program [116].

1.1 Motivation

Concurrency bugs widely exist in production software as is evident from the numerous bug reports and software related incidents. Concurrency bugs give birth to problems ranging from financial losses to the loss of human life in the case of safety critical systems [41, 63]. Some of the more well known concurrency bugs include a bug in NASDAQ's software systems which appeared during their initial public offering of Facebook and ended up costing NASDAQ over 13 Million US dollars in payments [115]. The northeast blackout bug in 2003 affected over 55 million people in parts of the United States and Canada, causing power outage for over two days [107]. A more deadly consequence of concurrency bugs can be traced back to the radiation overdose bug with Therac-25. The machine delivered a higher than prescribed dose of radiation to a number of patients and ended up causing severe injury, or in some cases even the loss of life [70].

Concurrency bugs such as data races, deadlocks, ordering bugs, and atomicity violations are notoriously hard to find. The difficulty in detecting concurrency bugs stems from the non-deterministic behaviour of concurrent applications. Concurrency bugs may appear only under rare circumstances, such as a rare interleaving of the program threads, and may require exercising every possible interleaving of threads in the program. Exercising every possible permutation of the interleaving between program threads to uncover concurrency bugs may turn out to be an infeasible problem for even small to medium scale applications. Ideally, programmers would like to test every possible state of a program, however in practice, it is infeasible and impractical to do so, if not impossible. The number of schedule permutations and consequently, the number of program states for even a seemingly simple program can be quite large in practice. Further, a program may have a large number of possible input and output combinations that follow different paths through the program and lead to different schedules of the program threads [87]. Attempts to unearth some anomalous behaviour in a concurrent program may in practice require testing all possible states in the program, which is a computationally intractable problem [76, 91]. The number of possible states at a program point increases exponentially with the number of threads executing at that point, as the memory operations in one thread may be interleaved with those in other threads in every possible permutation of the threads. In practice this permutation of thread interleaving can be astronomically large for any medium to large scale software system.

For an approach to be practical for detecting concurrency bugs, ideally, it must be able to deal with the non-deterministic behaviour of concurrent systems in a manageable way, i.e., without incurring a high overhead both in space and time. A practical approach must avoid or minimise reporting false positive alarms to the end user. Further, for large-scale software systems any practical approach must be able to identify a small part of the program code that exhibits the anomalous behaviour, for inspection.

1.2 Thesis statement

The primary objective of this thesis is to find effective techniques for identifying anomalies in concurrent software. Anomalies in software may be categorised as functional and non-functional anomalies. Functional anomalies are programming errors that affect the intended function of a software system and may cause it to crash or exhibit unintended and incorrect behaviour. Non-functional anomalies are those programming errors that affect the quality characteristics of a software system, such as the speed or responsiveness of the system. Depending on the properties and requirements of a software system, non-functional aspects like speed may also be treated as functional requirement (e.g., as is the case of hard realtime systems). Software anomalies need to be detected and eliminated early in the life of a software system, so that the system may perform its intended functions correctly.

The non-determinism that is inherent in concurrent software makes it difficult for programmers to reason about the behaviour of such applications. In a concurrent setting program threads may execute and interleave statements of the application in an arbitrary manner. Testing or exercising every possible interleaving of the program threads is not possible for any large scale real-world application. Program threads may interleave execution of statements due to a number of reasons — ranging from the affects of execution environment, current system load, to the affects of supplied program input. Application programs may display anomalous behaviour as a result of such a rare interleaving of the program threads, and it may be hard to reproduce the behaviour during debugging. Further, the non-deterministic interaction of the program threads in concurrent software makes it extremely challenging to find the source of anomalies in large-scale applications.

Most approaches in literature are limited in their scope of detecting the different types of concurrency bugs. This dissertation argues that concurrency bug detection techniques must not be limited in their scope to a certain type of bugs. A practical approach should be able to detect concurrency bugs regardless of whether the type of bug is a data race, an ordering bug, or an atomicity violation.

These considerations motivate the following thesis statement:

> *"Techniques for detecting rare anomalies in concurrent software need to account for the effects of non-determinism and should not be limited in scope to certain programming models, or environments. Such techniques are able to satisfy queries regarding dynamic behaviour of the software system without incurring significant performance overhead."*

1.3 Terminology

1.3.1 General terms

The dissertation makes use of the term programming error, anomaly, or bug interchangeably. All of the aforementioned terms refer to unintended program behaviour (either in the functional or the non-functional sense). The term unintended behaviour of a program is also used interchangeably with the term *deviant behaviour*.

Next, we present definitions for terms that are employed in this dissertation.

Definition 1 (Functional anomalies)**:** *In the functional sense an anomaly or bug is a programming error that produces unexpected results or causes the program to exhibit unintended behaviour (e.g., crash or fail). The following are examples of a functional bug or anomaly:*

- *A program is supplied with some specific input to produce a specific expected output as a result of the execution, but an output other than that expected is produced.*
- *A program crashes unexpectedly or fails to perform its intended operations as a result of some program input or interaction of the program threads.*

Definition 2 (Non-functional anomalies)**:** *In the scope of this dissertation a non-functional anomaly is a programming error that negatively affects the speed or responsiveness of the software system. An example of non-functional anomaly could be a program taking a large amount of time to perform a task that would otherwise take less time, if all other aspects remained the same (i.e., the program is supplied with the same input and execution conditions).*

Definition 3 (Program input)**:** *The term program input used in the context of this dissertation may refer to either supplied program inputs like a file(s) to compress, or a specific interleaving of the program threads, or a combination of both a supplied program input and an interleaving of the program threads.*

Definition 4 (Basic block)**:** *A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [2].*

Definition 5 (Program point)**:** *In the context of this thesis, a program point refers to a conditional jump in the program's control flow that marks the beginning of one basic block, and consequently, the end of another.*

Definition 6 (Bug manifestation point)**:** *We define bug manifestation point as a program point where the program's control flow path deviates from that of observed successful program executions, i.e., the program point where the bug has visible effect on program execution.*

Definition 7 (Deviant program point)**:** *A deviant program point is a measure of control flow similarity between executions of the program with the same program input. A program point that deviate from expected behaviour (i.e., the majority of executions) is referred to as deviant program point.*

Definition 8 (Program profile)**:** *A program profile records the number of times certain program events occur during a program execution [8]. For example, a statement-level profiler may count the number of times each statement is executed in program executions. Profiling may be performed at the level of a whole file, a procedure, a basic block, or even individual machine instructions.*

Definition 9 (Completeness)**:** *A debugging technique is considered complete if it does not report any false negatives, i.e., every true error is reported according to the domain of errors being addressed by the technique.*

Definition 10 (Soundness)**:** *A technique is considered sound if it only reports true errors, i.e., the reported error is confirmed by some observed execution behaviour of the program and is not a mere false positive.*

1.4 Summary and contributions

This dissertation identifies effective techniques for detecting anomalies in concurrent software, as outlined in the thesis statement. In pursuit of this objective, we present two different techniques for identifying functional and non-functional anomalies respectively. Further, we present the design of a framework to identify the root-causes of non-functional anomalies in concurrent software. The proposed framework builds upon and extends the techniques and ideas from our functional and non-functional anomaly detection approaches.

The central idea of this thesis is detecting anomalies by investigating anomalous program behaviour. In the context of this thesis program behaviour refers to properties of the dynamic execution of the program. These dynamic program properties include the program's control flow path, the CPU execution time consumed, as well as stall cycles and cache misses incurred in different parts of the program. The anomalous behaviour of program is referred to as *deviant behaviour* in the rest of this thesis. In essence, anything that is out of the "ordinary" is considered deviant behaviour. The key idea is that a program's dynamic execution behaviour reveals a representative set of its most common behaviour. A program's representative set of behaviour may be utilised as a reference for comparing against when the program behaves in a different manner, to determine whether the execution behaviour is deviant.

The approaches implementing the aforementioned ideas can account for effects of non-determinism and are not limited in scope to a single programming model or environment. The approaches are able to satisfy queries regarding the dynamic execution behaviour of the program being tested. The performance overhead of the online analysis for the approaches may not be low enough to be deployed in a production environment; however, it is not prohibitively high for in-house testing infrastructures and is in fact orders of magnitude times better than most state-of-the-art approaches in the literature (see Section 3.6.3 in Chapter 3 and Section 4.5.5 in Chapter 4).

This dissertation makes the following contributions to the state-of-the-art in anomaly detection:

- **Functional anomalies:** We present an infrastructure for detecting and localising functional anomalies in concurrent software. The infrastructure utilises dynamic control flow information from executions of the program to detect concurrency bugs in real-world applications. The approach can detect concurrency bugs that affect the program's control flow path, and can localise bugs to a small part of the program code. The approach does not require multiple failed executions of the program as is the case with most state-of-the-art approaches in literature. In most cases we can detect and localise software bugs by utilising just a single failing execution profile of the program. The approach offers a wider scope for bug detection and is agnostic to not only the types of bugs (i.e., data races, atomicity violations, ordering bugs, deadlocks), but also to the synchronisation model of the programming language used in the implementation of the application under test. Experimental results show that the approach suffers low run-time overhead as compared to other approaches in literature and it is effective in detecting the different types of concurrency bugs. The approach can cope with the effects of non-determinism that is inherent in concurrent software to prune out false positives.

- **Non-functional anomalies:** We present the details of an approach for detecting non-functional anomalies concerned with the speed and responsiveness of software systems. Experimental results demonstrate the effectiveness of the approach and show that the approach can detect previously unknown bugs in concurrent software. Fixing the reported bugs leads to orders of magnitude improvement in performance of the application. The approach can report about rare resource contention scenarios that would otherwise go unnoticed. The rare resource contention scenarios are identified by utilising multiple execution profiles. Analysing multiple execution profiles provides an overall picture of the resource consumption across a wide range of program inputs. The approach can filter out those parts of the program that are always slow on all observed executions and only reports dynamically observed deviant behaviour. The approach is precise and does not report any false positives.

- **Root-cause detection for non-functional anomalies:** We present the detailed design for a framework that can detect the root causes of non-functional anomalies in concurrent software. The proposed framework can detect anomalies by using the non-functional aspects of execution time, cache misses, and stall cycles. Once an anomaly is identified, the framework can also locate the root-cause for each of the identified non-functional anomalies in the software under test. We report the detected performance anomalies along with their associated root causes to programmers for verification.

1.5 Outline

The rest of this dissertation is organised as follows:

Chapter 2 presents an overview of the background knowledge necessary for a better understanding of the topics discussed in the later parts of the dissertation. The discussion involves definitions and descriptions of the different types of bugs plaguing concurrent software, as well as a discussion of the non-determinism inherent in concurrent systems.

Chapter 3 presents a framework for detecting and localising concurrency bugs. The approach works by detecting dynamic control flow changes in executions of the program. The approach can handle the non-deterministic behaviour of concurrent programs and can successfully prune false positives. The chapter presents results of the experimental evaluation of the approach, along with its shortcomings and scope of bug detection.

Chapter 4 presents a framework that introduces a profiling technique for detecting performance bugs in concurrent software. The approach can identify previously unknown performance bottlenecks in real-world software using a hybrid of dynamic and post-mortem analysis techniques. The chapter presents an experimental evaluation of the approach along with it's shortcomings and a discussion of possible extensions to the work.

Chapter 5 presents detailed description for a framework to identify the root causes of performance anomalies in concurrent software. The proposed infrastructure can detect performance anomalies and identify the probable root causes of such anomalies in parallel applications.

Chapter 6 presents a survey of the work related to this dissertation, including a discussion of the different approaches to concurrency bug detection in the literature. The chapter provides an overview of the different techniques for detecting both functional and non-functional anomalies along with the shortcomings of each approach.

Chapter 7 provides a short summary of the works presented in this dissertation. The discussion ends with some concluding remarks and future directions to extend the work of this dissertation.

2 Background

This chapter presents a discussion of the related concepts and background knowledge essential for a better understanding of the work presented in later parts of this dissertation. The discussion includes a high-level overview of the manifestation conditions of anomalies in concurrent software, and provides a description of non-determinism in practice.

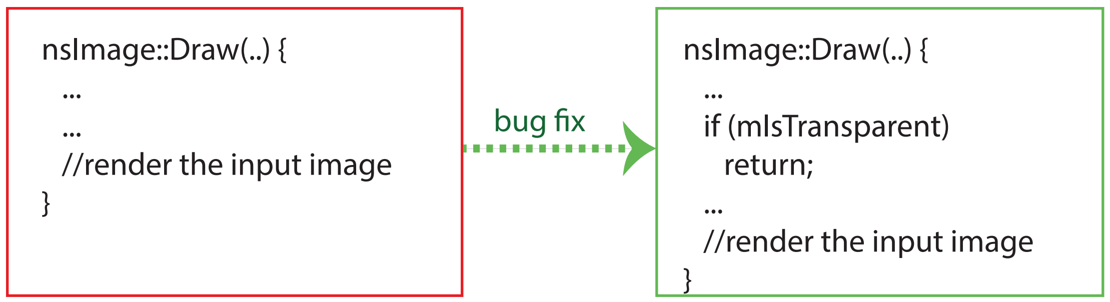
2.1 Software anomalies

In the context of this dissertation software anomalies (or bugs) are programming errors that cause applications to exhibit unintended behaviour, leading to either an incorrect result or degradation in performance. Programming errors may be introduced into software due to a number of oversights on the part of programmers or bugs in the compiler employed during software development. Mistakes that lead to programming errors include wrong assumptions about program behaviour, misconceptions about the problem definition, and errors in algorithm design or in selecting the wrong data structures [12]. Avoiding or correcting programming errors requires a good understanding of the problem statement and knowledge of the problem domain. Programmers must develop a clear understanding of the problem and design algorithms accordingly, while choosing the right data structures as dictated by the problem statement.

Software bugs may be classified as functional and non-functional anomalies. Non-functional anomalies are those programming errors that affect certain attributes or properties of the system. In the scope of this dissertation, these programming errors refer to anomalies that negatively affect the speed or responsiveness of a software system. As an example of a non-functional anomaly, consider a program that takes a large amount of execution time for some specific program input on some executions, but not all (i.e., the program completes its task in significantly less amount of time for the same program input on other executions). Figure 2.1(a) shows a simplified version of such a non-functional bug from Mozilla. [1]

The bug from the *Draw()* method for Mozilla, as shown in Figure 2.1(a), was

[1] Bug ID 6646 at https://bugzilla.mozilla.org/show_bug.cgi?id=66461



```
nsImage::Draw(..) {
  ...
  ...
  //render the input image
}
```

```
nsImage::Draw(..) {
  ...
  if (mIsTransparent)
    return;
  ...
  //render the input image
}
```

(a) Bug 66461: stretched transparent GIFs should not eat CPU time.

(b) Bug fix checks whether the image is transparent before rendering it.

Figure 2.1: Simplified version of a non-functional bug from Mozilla

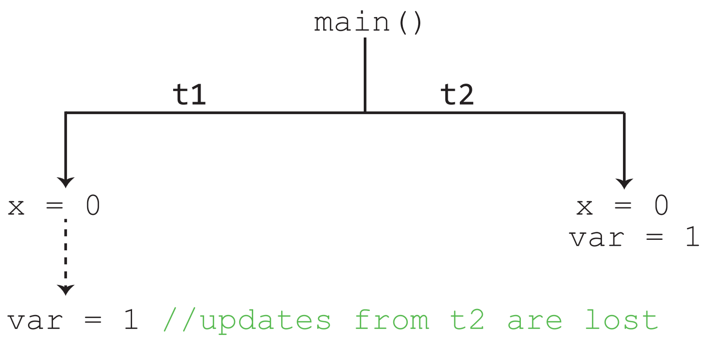
discovered when web developers started using transparent GIFs as spacers in their HTML documents. The *Draw* method performs the scaling, compositing, and rendering functions for images. These operations led to a waste of computation cycles for transparent images. The fix for this bug shown in Figure 2.1(a) is a simple check for images to see if they are transparent before initiating the rendering process, as shown in Figure 2.1(b).

Functional anomalies are programming errors that produce unexpected results, or cause the program to exhibit unintended behaviour (e.g., to crash or fail). As an example of functional bugs, consider that a program is supplied with some constant input, and it does not produce a predefined expected output, or the program crashes. The next section presents various cases of functional anomalies that plague concurrent software, including data races, atomicity violations, ordering bugs and deadlocks.

2.1.1 Concurrent anomalies

Concurrent software suffers from the same anomalies as its sequential counterpart. In addition, programmers writing concurrent software must also deal with the idiosyncrasies of various operating system schedulers, programming and memory models as well as hardware platforms. Anomalies in concurrent software may be further categorised according to the specific properties and conditions under which these bugs manifest.

Concurrency bugs are a result of unintended interactions between program threads that transition the program into an undesirable or incorrect state. Some concurrency bugs are a result of sequential thinking on part of the programmers, because it is hard to think about the states of multiple concurrent threads of execution. Concurrency bugs take on a number of different forms depending on the program state and manifestation conditions. The most common concurrency bugs include data races, atomicity violations, ordering bugs, and deadlocks. Next, we explain each of the aforementioned concurrency bugs and show the manifestation



```
static int var = 0; //Shared state

void run() {
   int x = var; // unprotected access to var
   var = ++x;
}

public static void main(...) {
   Thread t1 = new Buggy();
   Thread t2 = new Buggy();
   t1.start();
   t2.start();
}
```

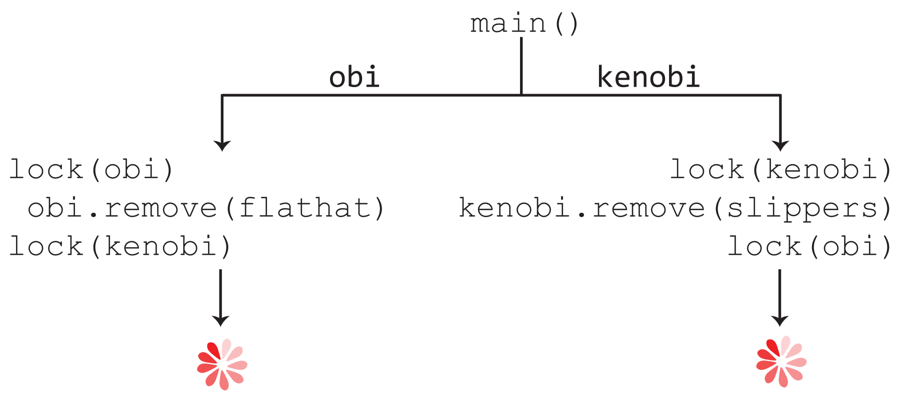
Figure 2.2: Simplified version of a data race

scenarios of each. Afterwards, we show how non-determinism works in practice using a simplified example.

Data races are concurrency bugs that result from uncoordinated access to shared memory in parallel programs. More specifically, a data race occurs when two or more program threads (or tasks) concurrently access the same shared memory location without holding a common lock or any other form of coordination and when at least one of the accesses is a write operation. Figure 2.2 shows an example of such a data race where the program threads are accessing the shared state *var* without any protection. The resulting program behaviour is non-deterministic, and depending on the order of execution of the threads, one thread might overwrite the updates from another thread.

Deadlocks result in a wastage of computational resources, and the program is unable to make progress. Deadlocks occur when two or more program threads circularly wait for each other to release the resources they have acquired. Figure 2.3 depicts such a situation where the two threads *obi* and *kenobi* will deadlock waiting for each other to release their respective locks. The program is stuck waiting forever, unable to make progress.

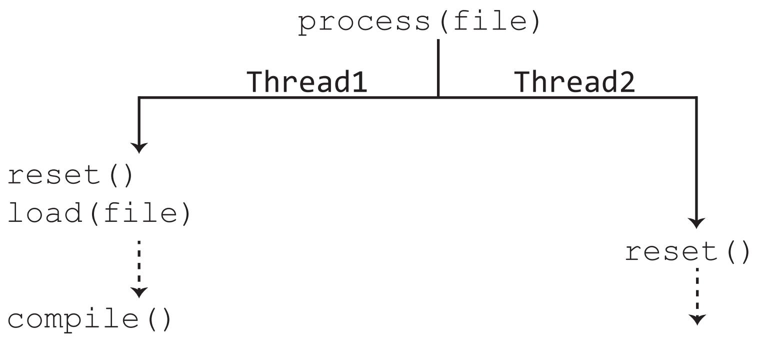
Atomicity violations are caused by an programmer's incorrect assumptions about non-atomic code regions to be executed atomically. A block of code in



```
Transaction {
   void transfer(to, item) {
      synchronized(this) {
         this.remove(item);
         synchronized(to) {
            to.insert(item);
         }
      }
   }
   public static void main(...) {
      Transaction obi, kenobi;
      ... //Thread 1
      obi.transfer(kenobi, "flathat");
      ... //Thread 2
      kenobi.transfer(obi, "slippers");
   }
}
```

Figure 2.3: Example of a deadlock

a program is atomic if for any parallel execution of the program, there is an equivalent execution with the same overall program behaviour in which the block is executed sequentially. Any interleaving of the program threads in these assumed-to-be-atomic regions will cause an atomicity violation. Figure 2.4 shows a simplified version of an atomicity violation. The functions are all synchronized (i.e., protected with locks), however, the program is still prone to behaving non-deterministically if the three function calls within *process()* are interleaved among program threads. In the given example (shown in Figure 2.4), *Thread1* gets scheduled first and performs the function calls to *reset()*, and *load()*; however, it gets de-scheduled before it can make the call to *compile()* the loaded file. At this moment *Thread2* gets scheduled and resets the *currentFile* pointer; however, it too is not able to make much progress, and *Thread1* gets rescheduled to complete its call to *compile()*. The function call to *compile()* within *Thread1* fails with an error because *Thread2* has already reset the *currentFile* pointer, and this causes the program to crash. Depending on the order of operation between the two threads,

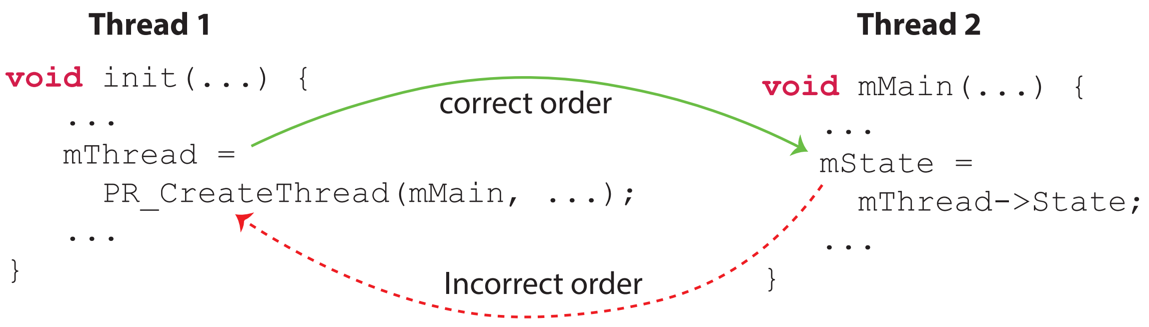


```
class ProcessFile{
     static File currentFile;
     synchronized void load(File file) {
          currentFile = file;
     }
     synchronized void compile() {
          //compile currentFile
     }
     synchronized boolean reset() {
          currentFile = null;
     }
     static void process (file) {
          reset();
          load(file);
          compile();
     }
}
```

Figure 2.4: Atomicity violation in the presence of synchronisation

the program will act non-deterministically and produce an unexpected result, even though the functions are protected with locks. In the given example, depending on how the operations are interleaved between program threads, the program behaviour may vary from expected behaviour, to a program crash, to compiling an altogether different file.

Ordering bugs are triggered when the execution order between statements executing in different threads is violated. In contrast to atomicity violations and data races, these bugs cannot be fixed by enforcing locks. The programmers in this case assume that one group of statements must always execute before the other group. An example of an ordering bug from Mozilla is depicted in Figure 2.5. Correctness of the program depicted in Figure 2.5 depends on the order of operations performed by the program threads. Executions that perform the operations in *Thread2* before those in *Thread1* lead to program failure because the program tries to access uninitialised memory.



```
void init(...) {
   ...
   mThread =
     PR_CreateThread(mMain, ...);
   ...
}
```

```
void mMain(...) {
   ...
   mState =
     mThread->State;
   ...
}
```

Figure 2.5: Simplified version of an ordering bug from Mozilla

2.1.2 Non-determinism in practice

Concurrent programs are highly non-deterministic in nature. A simplified version of a concurrent program is shown in Figure 2.6. The resulting non-deterministic behaviour of this program based on how the program threads get scheduled, is depicted in Figure 2.7.

Thread 1

```
...
x = x + 1;
x = x + 1;
...
```

Thread 2

```
...
x = x * 3;
x = x * 3;
...
```

Figure 2.6: Example of non-deterministic behaviour based on scheduling

The program in Figure 2.6 has two threads, both executing two instructionsprogra. *Thread1* increments the program variable *x* by one twice, whereas *Thread 2* multiplies *x* by the constant value of three twice. The resulting schedules shown in Figure 2.7 depicts that depending on how the program threads are scheduled/interleaved, the program may exhibit different behaviour on different executions of the program. In Figure 2.7 the solid bars represent execution (i.e, the thread is currently executing program instructions), whereas, the dotted bars represent inactivity (i.e., the thread is currently waiting in the scheduling queue). The numbers on top of the solid bars represent the resulting values for the shared field *x* as the two threads execute the program. The last column shows the final output of the program, depending on the different schedules/interleavings of the program threads. Understanding and/or debugging the resulting thread schedules for programs with such non-deterministic behaviour becomes especially troublesome in large-scale software with millions of lines of code, executing a large number of program threads.

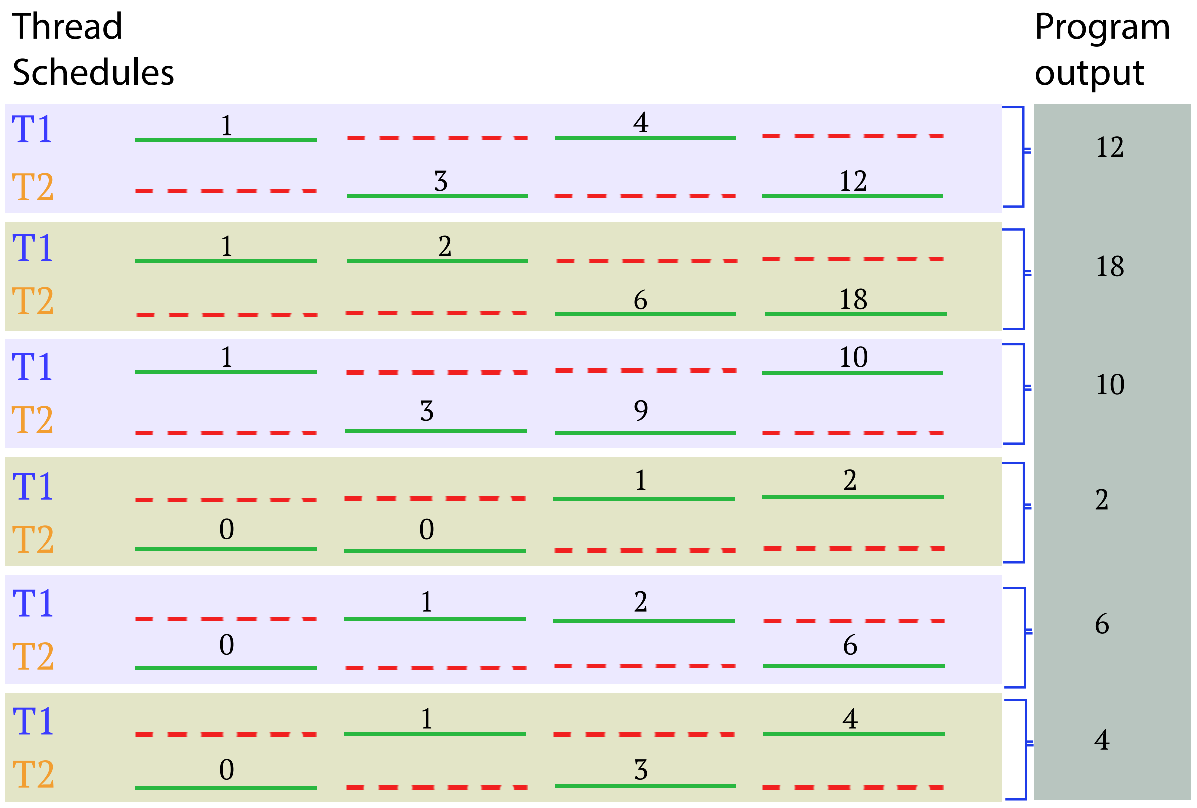


Figure 2.7: Representation of non-determinism in concurrent programs

2.2 Related concepts

2.2.1 Dynamic analysis

The approaches presented in this dissertation are based on dynamic analysis of the runtime behaviour of client programs. Dynamic analysis is concerned with analysing the runtime properties of a program as it executes to detect violations of certain properties as well as to provide information about the behaviour of the program. Dynamic analysis derives program properties that hold for one or more executions of the program by examining or recording the runtime behaviour. Tools or approaches that employ dynamic analysis usually make observations about properties of the running program by employing program instrumentation [8]. Instrumentation adds code to a program. The added code may sometimes refer to instrumentation code as well as analysis code. Instrumentation code is a set of instruction for deciding on when and where to instrument/insert analysis code. Analysis code is the set of instructions inserted by the instrumentation code at predefined instrumentation points in the client program. Instrumentation may be performed either statically (i.e., without executing the program) or dynamically (i.e., as the program executes). Further, instrumentation may be performed either at the source code level or at the binary level. Binary instrumentation involves analysing programs at the level of machine code, and analysis is performed at the level of executable intermediate representations such as byte-codes or specific instruction set architectures.

The program properties observed or recorded by using dynamic analysis provide precise analysis information. The information is precise because instrumentation can be tuned to collect only the information required for the problem domain. The program properties derived via dynamic analysis are true for the observed set of executions; and may not hold for all executions of the program. However, dynamic analysis does not suffer from the problem of infeasible paths (paths that can never execute), as is the case with static analysis, because dynamic analysis examines actual program executions. On the other hand, dynamic analysis covers fewer execution paths through the program as compared to static analysis because the information collected via dynamic analysis is input-centric in the sense that the properties derived with dynamic analysis are dependent on the program input being used and may change if the program is executed with a different input.

One of the major concerns with dynamic analysis is the performance overhead incurred by client programs. Since dynamic analysis must make on-line observations about the runtime behaviour of the program, the performance overhead incurred by applications depends on two pieces of detail. First, the overhead is dependent on the granularity of the analysis and instrumentation routines. For example, collecting analysis information about every instruction of the program may incur a higher overhead compared to collecting information at the higher granularity of an entire procedure. Second, the overhead associated with the instrumentation framework being employed for analysis also makes considerable contribution to the overall performance penalty suffered by the client applications. Generally, the client program must be executed under the control of the instrumentation framework, which implies that the client program must be loaded into the framework. The instrumentation framework typically intercepts the execution of client executables and generates new code to be executed in a just-in-time fashion. The original client code is usually not executed and is used only for reference. The process of intercepting client instructions and generating new code contributes to the performance overhead.

This dissertation employs binary instrumentation to perform dynamic analysis of the client programs. Dynamic binary instrumentation has the advantage that it usually does not require altering or re-compiling the client program and does not require access to program source code. The instrumentation and analysis tools presented in this dissertation are based on Pin [77], which is a dynamic binary instrumentation framework for IA-32 and x86-64 instruction set architectures. Pin tools insert code written in C/C++ into arbitrary places in the program binary. The analysis code is injected into the client program at runtime by the instrumentation code that is grafted onto the client program. The code is added dynamically as the program executes.

2.2.2 Post-mortem analysis

The frameworks presented in this dissertation perform post-mortem analysis of the recorded dynamic behaviour of client programs for the observed executions. Post-mortem analysis is a secondary step that follows after a client program has finished performing some task. The analysis involves utilising the recorded information about dynamic program behaviour as observed during program execution(s). The observed/recorded information about dynamic properties of the program serves as post-mortem evidence of the program behaviour. The post-mortem analysis performed as part of the infrastructures presented in this dissertation consists of offline analyses, i.e., they don not affect the execution or performance of the client program. The idea is that rare concurrency bugs are hard to reproduce, so once a failure has occurred, post-mortem analysis can deduce the actions that led to the error, without the need for re-execution to reproduce the error.

2.2.3 Hardware performance counters

Part of the work that we present relies on the use of hardware performance counters for obtaining statistics about the execution behaviour of client programs. Hardware performance counters are a set of special-purpose registers built into most modern microprocessors. These special-purpose registers are dedicated to monitoring and recording various hardware-related events such as cache misses or taken branches. The number of special-purpose registers is usually small; however, these registers may be programmed to monitor any of the large number of supported hardware-related events at runtime. A common counter, for example, is the number of instructions committed since the counter was enabled. When these counts are made available to software, they can be used to find and remove software inefficiencies [125]. Most modern processors, including all x86 CPUs, are equipped with hardware performance counters. Accessing information regarding a counter involves polling the pre-configured counter register periodically, or during the handling of a counter-overflow interrupt. Accessing hardware performance counters incur low overhead since performance counters are provided at the hardware level.

There are a variety of software tools that provide high-level interface and analysis capabilities for performance monitoring hardware. The Performance Application Programming Interface (PAPI) tool suite [126] provides a common interface to performance monitoring hardware for many different processors including PowerPC, Pentium, MIPS, Cray, and Itanium. Similarly, Intel’s VTune Performance Analyzer [112] supports the performance monitoring hardware of all Intel Pentium and Itanium processors.

3 Detecting and localising concurrency bugs

In this chapter we introduce an approach for detecting and localising concurrency bugs. The approach works by detecting control flow changes in program executions. The idea is that if the input is fixed, then the control flow path taken by the program should not change for executions of the program under different interleavings of the program threads. The approach can handle the inherent non-deterministic behaviour of concurrent programs and can successfully prune out false positives.

3.1 Overview

The non-deterministic nature of concurrency bugs makes it challenging to find and correct these bugs. One reason is that such bugs may be triggered under rare circumstances, such as a low probability interleaving of the program threads. Once a failing execution of the program is observed, ideally we should be able to decipher from that failing run, a location in the program where the bug manifests, i.e., makes an observable impact on the behaviour of the program.

In this chapter we present an approach for detecting rarely occurring concurrency bugs and show how to localise them to a small part of the program code. The key idea is that buggy executions may deviate from the control flow path for correct executions. Our approach can detect this deviation from expected behaviour by identifying Deviant Program Points (DPPs). The approach employs dynamic binary instrumentation to record the number of conditional branches that are entered over different executions (while keeping the same input). Binary instrumentation provides a reliable way to monitor the execution of conditional branches. This information is mapped back to the source program to allow the user to understand the detected problem in source language terms. When applied to real-world concurrency bugs, our tool can pinpoint the location(s) in code where program's behaviour deviates from that of a correct execution. Our approach eliminates false positives and localises a deviant program point with a high level of accuracy by using a single failed program run. Our approach is orders of magnitude faster than state-of-the-art detection techniques for concurrency bugs.

3.2 Introduction

Concurrency bugs have been consistently surfacing in real world software for more than three decades. These bugs go back as far as the deadly Therac 25 radiation overdose bug in 1985 [70]. Some more recent examples of concurrency bugs include the famous Northeast blackout bug in 2003 [107] and a bug in the IPO software of NASDAQ's that appeared during their initial public offering for Facebook in 2012 [115]. Each of these bugs has had serious consequences, ranging from the loss of human life, to over 55 million people remaining without electricity for days, to a $10 million penalty. In a survey conducted at Microsoft to assess the state of the practice of concurrency [44], two-thirds of the respondents reported that they had to deal with concurrency issues and/or fix bugs on at least a monthly basis. It is further reported that fixing these bugs took at least a few days — accounting for "thousands of days of work" in aggregate [101]. This problem has been considerably exacerbated by the widespread adoption of concurrent and multi-threaded programs. Multi-threaded programs are needed for efficient utilisation of the multi-core processors of today. These processors are not limited to high-end servers and desktop machines anymore, but are also present in most modern handheld devices like smartphones. The widespread use of multi-core processors is enforcing the need for writing concurrent software. As a result, concurrent software is becoming more pervasive, however, writing concurrent programs is hard because programmers are used to sequential ways of thinking. Programmers frequently assume non-atomic code to be executed atomically without explicitly enforcing it, which results in buggy code.

Detecting concurrency bugs is challenging, but it is equally hard to localise them, i.e., to identify the part of program code where the bug manifests itself or has a visible effect on the execution of the program. Fixing a bug is not straightforward, either. Once a bug is "fixed", it is hard to ascertain if the right bug has indeed been resolved. This uncertainty arises because the programmer could accumulate false confidence in the system after observing only successful program runs. The non-deterministic nature of concurrent programs makes it difficult to repeat these bugs during diagnosis. Ideally, we would like to make sure that once a bug is triggered, we can identify those locations in the program where the bug affects its execution. Afterwards, these bug-affected program points may be further analysed to find out the root cause of the bug.

The approach presented in this chapter is based on the strategy of identifying deviant program points. That is, once a buggy program execution is observed, it is possible to compare it to a correct execution of the program and to figure out the program point(s) where the buggy execution deviates from the expected behaviour. However, merely identifying differences in execution are usually not enough. These differences in execution could be false positives because of the non-determinism prevalent in concurrent programs. Narayanasamy et al. describe it a challenging activity to eliminate false positives. They report that only around 10% of data races are harmful and could lead to software failures. The remaining

90% are false positives and must be eliminated to avoid overwhelming developers with false alarms [90]. Furthermore, any system that aims to detect and localise concurrency bugs must ensure two properties: First, such a system must minimise the number of reported false positives; And second, the amount of performance overhead must remain within reasonable bounds. Most prior concurrency bug detection tools either have high overheads [114], or require specialised hardware to circumvent the high overheads [149].

As a motivating example of a concurrency bug, we present the simplified version of a reported bug in Pbzip2 version 0.9.4. The bug is shown in Figure 3.1. We later show (in Section 3.6) how this bug can be detected and localised with our technique. In Pbzip, the program spawns consumer threads to perform compression, and it spawns output threads for writing the compressed data to file. The main thread issues a *join()* for all output threads, but at Line 13 Figure 3.1 it does not issue a join for any of the consumer threads. When the main thread frees resources, it is possible that some consumer threads are still not done (Line 5 in Figure 3.1). This means that on Line 18 in Figure 3.1, when the main() function frees *fifo→mut* by invoking the *queueDelete()* function, some of the consumer threads may be still active. If these active consumer threads attempt to execute the *fifo→mut unlock* statement, the program crashes with a segmentation fault.

3.2.1 Background

Concurrency bugs most frequently manifest as atomicity violations, data races, ordering bugs, or deadlocks. Each of these classes of concurrency bugs has distinct properties that must be exploited to detect and localise them. A large body of work has been dedicated to static and dynamic approaches for exposing concurrency bugs. Some of the early work on detecting data races includes the lockset and happens-before based approaches [96, 114]. The lockset-based approaches and their happens-before counterparts maintain and dynamically update a set of locks that are required to access each shared memory location in the program. Zhou et. al propose a hardware implementation of the lockset-based approach in their tool HARD [149], in which they use bloom filters to circumvent the high overhead associated with lockset-based approaches. However, the need for specialised hardware, and a the problem of reporting a high number of false positives as well as false negatives, remains a problem due the nature of bloom filters and the lockset-based approach. Others have focused on detecting deadlocks [32, 60], while recently the focus has shifted towards detecting atomicity violations. Early work on detecting atomicity violations relies on the developer to explicitly annotate atomic code regions to help the approach identify faults either statically or dynamically [38, 40].

The work presented in this chapter does not suffer from the aforementioned shortcomings. First, our approach does not require special hardware, neither is it plagued by false positives. Second, our work does not require any explicit annotations by the user — the only value that the user may supply is a pruning

```
void main(...) {
        ...
        for (i = 0; i < numCPU; i++)  {
                ret = pthread_create(&con, NULL, consumer, fifo);
                ...
        }
        ret = pthread_create(&output, NULL, fileWriter, OutFilename);
        ...
        // start reading in data
        producer(..., fifo);
        ...
        // wait for all output threads to finish
        pthread_join(output, NULL);
        ...
        fifo->empty = 1;
        ...
        // reclaim memory
        queueDelete(fifo);
        fifo = NULL;
}
```

```
void *consumer(void *q) {
        for (;;) {
                pthread_mutex_lock(fifo->mut);
                while (fifo->empty) {
                        if (allDone == 1) {
                                pthread_mutex_unlock(fifo->mut);
                                return NULL;
                        }
                        ...
                }
                ...
        }
        ...
}
```

Figure 3.1: An atomicity violation bug in pbzip2-0.9.4.

threshold value used for eliminating false positives. [1] And, finally, the approach is not limited to a single type of concurrency bug, but it can detect any bug that causes changes in the program's control flow path.

A precondition for detecting and localising concurrency bugs is that the bug must be triggered. Furthermore, two conditions must be met to trigger a concurrency bug: First, a bug-triggering input must be supplied, and, second, a bug triggering interleaving must be forced in the program. The first condition of bug triggering input is not unique to concurrent programs. Generating Bug triggering input has been the subject of ample research [98, 99, 118]. These approaches are able to generate a wide range of input for testing corner cases and provide better code coverage. The second condition of forcing bug triggering interleavings is unique to concurrent programs. The approach generally employed to provide

[1] For details about the pruning threshold value see Section 3.4.2.

interleaving coverage is to control access to shared memory and/or control how the threads are scheduled at different points in the program. Both controlling access to shared memory and thread scheduling may be performed in a random or systematic manner [16, 30].

Generally, when a bug is triggered or identified developers must manually localise the bug to identify its source in the program [58, 101]. Other approaches try to relieve developers of this burden by logging and analysing data to automatically localise these bugs. The analysis of logged data includes: looking for recorded branch outcomes [58], or bug patterns [101]. These approaches then assign scores to the detected predicates/patterns depending on how often the predicates/patterns appear in failed and successful program runs. To perform their ranking of suspiciousness, these approaches must observe and record a high number of failed program executions beforehand. Our approach, on the other hand, is able to detect and localise bugs as soon as a single failed execution of the program is observed.

Some of the more recent work on detecting and localising atomicity violations and ordering bugs take a step further. These approaches try to record the exact memory accesses that are involved in the manifestation of the bug. The recorded potential bugs are then assigned a statistical rankings of suspiciousness in the case of Falcon [101], whereas, in the case of LOCON [132] an attempt is made to match the potential bugs against a set of bug patterns. In the case of LOCON three different test procedures are executed to ensure that both single and multi-variable atomicity violations are eliminated. Both these techniques monitor all shared memory accesses that are performed in a program online to detect and record conflicting access patterns. The monitoring of all accesses to shared memory in these approaches results in a runtime overhead that is directly proportional to the size of the program being tested.

Our approach circumvents the high overhead incurred by Falcon and LOCON by recording only the control flow decisions taken by the program instead of monitoring and recording every shared memory access. The approach does not require multiple procedures to detect the different types of concurrency bugs. Our use of control flow information has the advantage of low overhead and enables our approach to detect any concurrency bug that may change the program's control flow path, as opposed to a single class of bugs.

3.2.2 Contributions

Generally with concurrent programs it is hard to collect many failing executions of the program for rarely occurring bugs. These rare bugs only appear under certain low probability interleavings of the program threads. Therefore, once a failing execution of the program is observed, ideally we should be able to decipher a location in the program, where the bug actually manifests. To show that this is a real problem, we point to two Mozilla bug reports and a bug in the Apache httpd server. In the first instance we look at Mozilla bug report #73291, where developers initially gave up on the bug and closed the bug report because they could not reproduce the bug, while in another instance (bug #72599), developers gave up on repeating the bug and simply submitted a patch based on their "guessing" [76]. Similarly, for an atomicity violation bug from Apache web server presented in [102], the authors report that it took 22 hours on an 8-core machine for the bug to manifest, while running Apache with a bug-triggering input.

The technique presented in this chapter attempts to eliminate the need for this aforementioned guesswork by identifying and reporting the exact point(s) in the program where execution deviates from the behaviour of a correct program execution. The technique is able to accurately identify sub-conditionals inside a potentially complex conditional that is responsible for the deviant behaviour of the program. We show in Section 3.6 with the bug from StringBuffer presented in Figure 3.8, how our technique is able to precisely assign blame to the sub-conditional *(srcEnd < 0)* at Line 21 for the bug. Our technique is able to identify these sub-conditionals even when a compiler implements short-circuited booleans - as is the case with most modern compilers. We refer to the identified program points as *Deviant Program Points*. Our definition for a Deviant Program Point (DPP for short) is presented in Section 1.3.1 in Chapter 1. DPP refers to a program point that is not observed during all of the recorded executions of the program. The combination of whether a conditional branch is taken during execution, along with the number of times that the branch is taken, is considered in conjunction when reporting a program point as deviant.

Most of the statistical approaches in literature [58, 101, 102] require observing/capturing multiple failed executions of the program to facilitate their ranking or sampling mechanisms. Current approaches for detecting and localising concurrency bugs focus on a single type of concurrency bug or have multiple procedures that need to be separately executed for detecting a small subset of these bugs [101, 102, 132]. In contrast, the technique presented in this chapter can detect and localise any bug that changes the program's control flow path, offering a wider scope. Furthermore, our technique requires only a single failed execution to find the DPPs as opposed to the need for capturing multiple failed runs of the program.

In summary, this chapter makes the following contributions:

- We present a dynamic approach to detect concurrency bugs by identifying Deviant Program Points (DPPs). The DPPs are used to localise bugs to a small part of the program code where the bugs manifest. The approach requires a single failed execution to identify DPPs and is agnostic to not only the kind of bugs but also the synchronisation model being used - offering a wider scope of bug detection.

- Our approach can eliminate false positives by accounting for non-determinism that is inherent in concurrent programs, load balancing, as well as filtering out DPPs that point to *libc* and other libraries that are outside the scope of the program being tested.

- The approach performs orders of magnitude faster than the most recent approaches for concurrency bug detection and localisation. It also scales better to larger programs, since it only monitors the conditional branches that are executed in a program.

- Lastly, we present a prototype implementation of our approach to demonstrate its effectiveness and scalability. We present an evaluation using a number of micro benchmarks, as well as bugs in Aget, PBZIP2, Apache httpd web server, and an implementation of StringBuffer from jdk 1.4.

3.3 Bug detection and localisation conditions

The technique presented in this chapter views bug detection and bug localisation as separate, but related issues; with localisation being dependent on detection. That is to say, that a bug must be triggered and consequently detected first, before it can be localised to a part of the code that is responsible for the bug.

3.3.1 Bug triggering

The approach is primarily concerned with detecting concurrency bugs, and not triggering them. Bug triggering may be achieved by executing the program against input supplied by the user, or supplied by a separate program to generate test inputs. There is extensive literature on test generation [16, 31, 72, 81, 99, 110], but the topic is beyond the scope of this chapter. However, to help the scheduler induce a degree of non-determinism in the program, the approach may delay random accesses to shared memory in the program under test. The delays are caused by injecting *sleep* into the program binary for a random amount of time (in microseconds). Delays are meant to influence the scheduler and increase the chance of a rare interleaving of the program threads to occur our system. The program binary is instrumented at runtime to inject delays.

3.3.2 Bug detection

The approach is able to detect bugs by executing a program multiple times and observing/recording changes in the control flow path that is followed by the program during the current run as compared to a previous run (either failed or successful). The program input for the recorded executions must remain the same for the approach to work. In the context of our approach, If the program input is fixed and the effects of any non-deterministic behaviour (e.g., reading a random value, or time of day) on control flow are eliminated, then, any changes in the control flow path of the program may point to a potential bug.

3.3.3 Bug localisation

Localising a bug that is detected using our approach is achieved by identifying the exact deviant program point(s) where the detected bugs manifest [2]. Bug localisation identifies the locations inside program code that are pointed to by the identified DPPs.

Both bug detection and bug localisation requires at least one failed and one successful execution of the program to work. The list of DPPs is compiled by comparing branch decisions taken during successful and failed program runs. Multiple DPPs may be identified as potential bugs, based on any changes that may be observed in the control flow path of the program. However, not all of the DPP may represent a bug in the program. A DPP may be the result of some inherent non-deterministic behaviour of the program. For example, the program might make control flow decisions based on reading a random value, or performing operations based on the size of some buffer. This non-deterministic behaviour of the program may lead to reporting potential bugs that are false positives.

3.3.4 Bug replay

The approach is not concerned with replaying/reproducing concurrency bugs. There are numerous approaches in literature for record and replay frameworks [23, 54, 84, 89, 103] that may be used for this purpose. In general, record and replay tools are needed to find the exact location of a bug, however, these approaches are orthogonal to our technique. The reason that our approach does not require replaying or reproducing a bug that is triggered is that once a bug is triggered, it is recorded in the logs generated by our dynamic analysis and becomes part of the record. The bug record/log may then be utilised in any way that the offline analysis requires.

3.3.5 Eliminating false positives

The elimination of false positives is achieved by collecting and observing the execution behaviour of other program runs (while keeping the same input). These further program runs may be either successful or failing ones, depending on which one is more deterministic to reproduce. That is to say that we may run into cases such that: a bug is highly non-deterministic, and as a consequence a failing program execution of the program is observed rarely compared to a successful one, or the other way around for when bugs are deterministic. In the first case we may need one failing execution profile and a few[3] successful runs to be able to eliminate most false positives. In the later case, the situation is reversed, and we may need one successful and a few failing program runs. In either of the aforementioned

[2]Note the definition of bug manifestation from Section 3.2.

[3]In our experiments with a bug in Aget, four successful execution profiles against one failed one were enough to localise the bug with only five false positives.

cases our goal is to detect deviations from the "normal", i.e., expected behaviour of the program. A distinction between failing and successful program runs is not required by our approach, since the approach relies on detecting differences in the program's control flow path. The downside is that one may perform the analysis without recording any failing program run and miss a bug, however, the situation may be avoided if test cases to trigger a failure are available.

3.3.6 Scope of bug detection

The approach presented in this chapter offers a wider scope as compared to earlier approaches in literature that are limited in scope to a single type of concurrency bug [16, 101], or have multiple modes (or procedures) [132] that must be separately executed to detect each type of bug. The approach is able to detect any bug that causes a change in the control flow path followed by the program. If an execution (successful or failed) takes a different control flow path (or takes only a portion of the path - in the case of crashes), then our technique will detect this and identify the DPPs involved in the anomalous behaviour. Table 3.1 on Page 38 lists the different categories of bugs that the tool has been able to detect and report successfully.

3.4 Methodology

Our approach works in two phases. The two phases consist of dynamic analysis of the program at runtime and an offline post-mortem analysis of the profiled data. The high-level architecture of the approach is presented in Figure 3.2. The analysis phases involved are explained in more depth next.

3.4.1 Online phase

In the first phase, the program to be tested is executed multiple times under control of our tool to try and capture a failing program run. Our system logs and keeps a record of all the conditional branches that are taken by the program during execution for both the successful and failing runs. Once a failing run is observed — either by a human operator or a test case, the process may be stopped. Our system keeps track of the execution counts for all conditional branches that are taken during program execution by each thread. The final execution count for each conditional branch is calculated by summing up the individual execution counts from each program thread. The reason why it is important to calculate the sum total is to deal with issues of load balancing within the application. The observation is that load balancing in an application could result in different numbers of thread to execute within a program in different executions. This implies that under different system loads, a different number of threads may attempt to execute a conditional branch. Therefore, for a conditional branch the individual execution counts for threads may be inconsistent in different program runs. The key idea is that if the

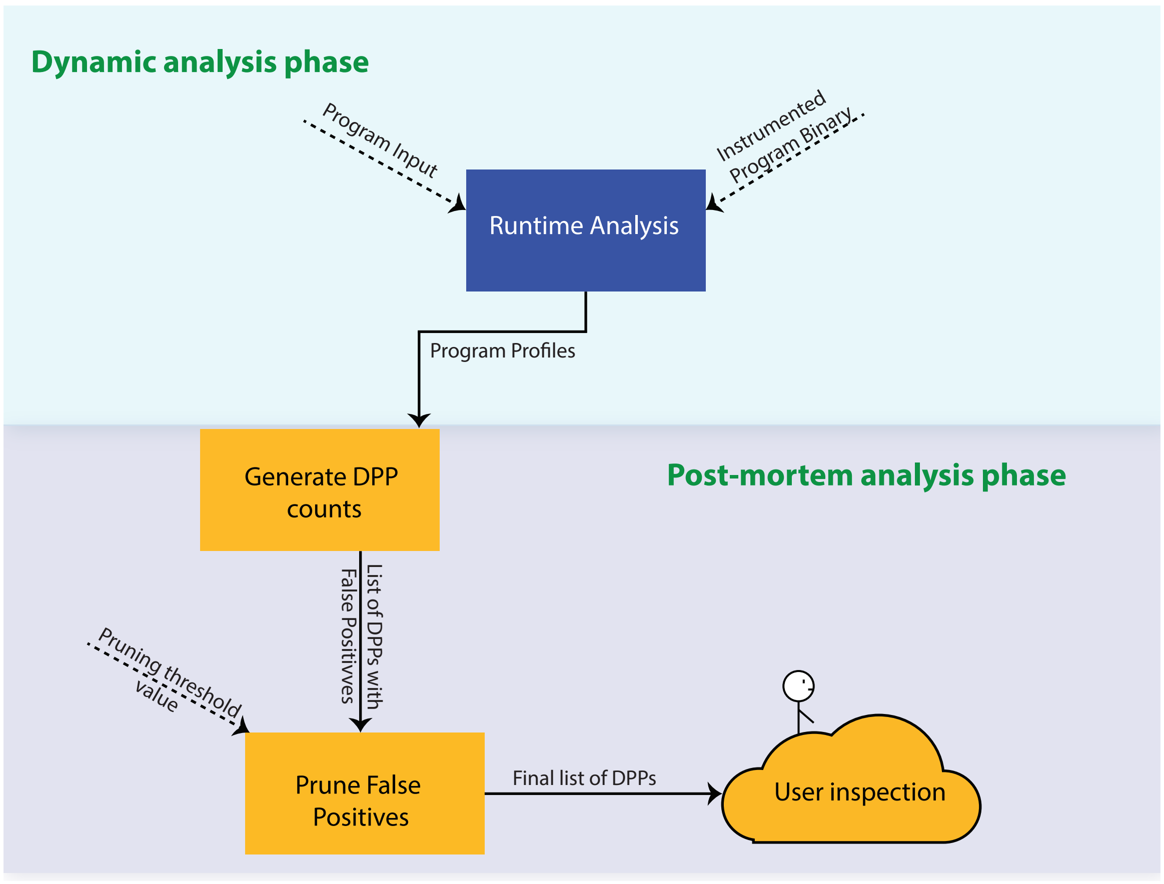


Figure 3.2: High-level overview of the system architecture.

program input is fixed and the execution is deterministic based on input, then the total number of times that a branch is executed (entered) must remain constant [4]. For the observation to hold in reality, no changes to user or network input must be tolerated, since these may cause changes to the control flow path of the program. However, in practice program executions are not always deterministic based on input, which may lead our technique to report false positives. The procedure to eliminate false positives is laid down in Algorithm 2 and further explained in the later part of this section. Every execution of the program creates an *execution profile* of the program being tested. The execution profile for a program contains individual execution counts for each program point (i.e., conditional branch) in the program that gets executed. Once our system has observed at least one failing execution of the program, the process may be stopped to start the second phase.

3.4.2 Post-mortem analysis phase

The second phase of our approach is concerned with performing a post-mortem analysis of the data collected in the online phase. In this phase a list of DPPs is compiled from the recorded execution profiles. Algorithm 1 shows how the list of DPPs is generated from the execution profiles recorded in the first phase. The list of

[4]The situation with spin locks or signal/wait synchronisation is addressed in the section on pruning algorithm on Page 33.

```
input  : $eProfiles$ - List of execution profiles for a single program input
         loaded from disk
output: $fSet$ - A map of DPPs containing false positives
while eProfiles has more elements do
    $currProfile \leftarrow eProfiles.next()$ ;
    for record in currProfile do
        if record.pPoint $in$ fSet then
            $count \leftarrow record.pPoint.count$;
            if count $in$ fSet.pPoint.brCountSet then
                $fSet.pPoint.count.freq++$ ;
            end
            else
                $fSet.pPoint.add\{count : 1\}$ ▷ Execution Count:Observed frequency
            end
        end
        else
            $fSet.add\{pPoint : \{count : 1\}\}$;
        end
    end
end
return fSet;
```

Algorithm 1: Generating the list of DPPs from execution profiles.

DPPs is generated by comparing the control flow path followed by the program in a failing execution of the program with the control flow path(s) followed in successful execution(s). The comparisons are performed on execution profiles collected for a single program input. A single DPP in the list corresponds to a program point that appears in only some but not all execution profiles of the program. The list of compiled DPPs represents the changes that are detected between the control flow path (i.e., execution profile) for failing program executions compared to successful ones. The list of DPPs may be compiled with or without any pruning of false positives (explained below). Finally, a list of DPPs is reported to developers that may be further inspected to ascertain whether they are true bugs.

Pruning phase

The second phase our technique may include an optional pruning sub-phase. This sub-phase uses a pruning threshold value.

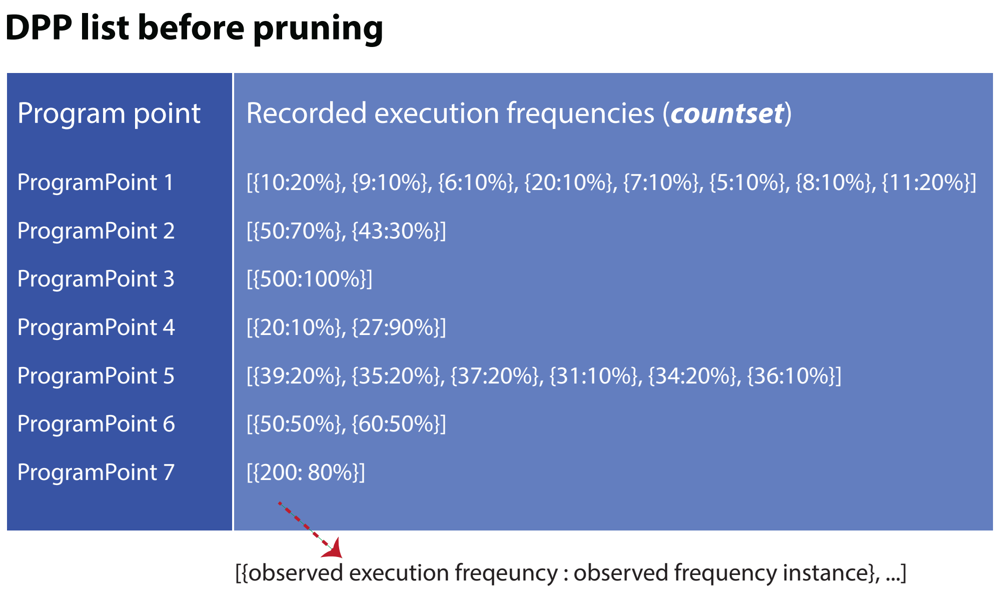
Definition 11 (Pruning Threshold value)**:** *A pruning threshold value is used for removing those DPPs from profiled data that our approach deems uninteresting to report. All those DPP for which the execution frequency has a high variance, or whose execution frequency is above the specified threshold are uninteresting to our approach. A high pruning threshold value implies less pruning, while a lower value implies aggressive pruning of the profiled data. The default pruning threshold* *value of 5% for our technique eliminates all entries that are common to 95% of the recorded executions.*

The pruning threshold value is used for eliminating false positives from the list of DPPs collected during the first phase. Figure 3.3 shows an example of the pruning process for 10 executions of the program (the *n* value). Figures 3.3a to 3.3c show the effect of choosing a low (30%) and high (90%) pruning threshold value. In 3.3b a threshold value of 30% is chosen instead of the default 5% because of the small *n* value. In Figure 3.3b *ProgramPoint1* and *ProgramPoint5* are removed because their execution counts are highly non-deterministic (by Line 8 of Algorithm 2), while *ProgramPoint3* and *ProgramPoint6* are removed because their execution frequency is highly deterministic (by Line 12 of Algorithm 2). *ProgramPoint7* is a special case, where the compiler reverses the conditional, so that *not* executing the instruction is buggy behaviour. An example of a situation where the compiler reverses the conditional is the code snippet shown in Figure 3.5 on Page 34. Compiling the program code in Figure 3.5 with the *g++* compiler from the GNU Compiler Collection (version 4.8.2) produces identical assembly jump instructions for both of the *if* conditionals in the two threads, although the conditional in *Thread2* is a negation of that in *Thread1*. The generated code performs a logical exclusive disjunction of the flag being tested in *Thread2* with the constant number one (the value for logical *true*) to make the code behaviour identical. As a consequence, this bug only appears in the list of reported DPPs if we choose a high pruning threshold value. In Figure 3.3c a high pruning threshold value 90% is chosen, i.e., only the most deterministic branches are eliminated and everything that is non-deterministic in the slightest is left out. The problem with choosing a high pruning threshold value is that we may end up reporting false positives. In the case of Figure 3.3c, the report will include the probable false positive referring to *ProgramPoint6*, since the execution counts for the program point appear to be deterministic (i.e., it does not refer to a rare bug).

The Pruning threshold value is used to eliminate both highly deterministic and non-deterministic false positives from the list of compiled DPPs.

All DPPs or their individual execution counts in the *countset* that are above the pruning threshold value are eliminated. The *countset* for a DPP is a set containing the different observed execution counts for that DPP and the number of times that the executions counts appear in the recorded executions profiles. The default pruning threshold value for our technique eliminates all conditional branches that are common to 95% of the recorded executions. The pruning threshold value may be optionally specified by developers using the system. After performing pruning, the final list of DPPs is compiled and reported to the user for inspection. The details of our pruning algorithm are shown in Algorithm 2.

Pruning algorithm: Pruning of execution profiles is performed to eliminate any false positives that may appear in the full list of DPPs compiled in Algorithm 1. False positives are a consequence of the non-determinism inherent in concurrent programs. While performing pruning of false positives, all the DPPs that are

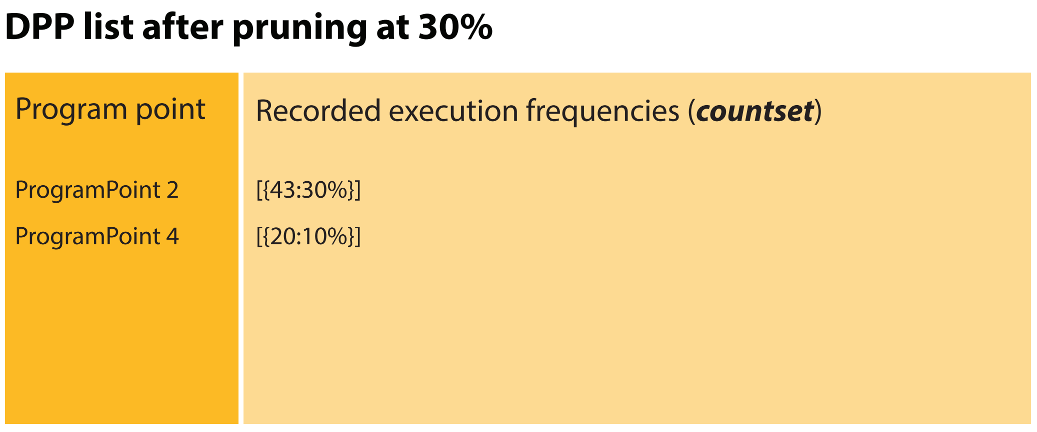


DPP list before pruning

Program point	Recorded execution frequencies (***countset***)
ProgramPoint 1	[{10:20%}, {9:10%}, {6:10%}, {20:10%}, {7:10%}, {5:10%}, {8:10%}, {11:20%}]
ProgramPoint 2	[{50:70%}, {43:30%}]
ProgramPoint 3	[{500:100%}]
ProgramPoint 4	[{20:10%}, {27:90%}]
ProgramPoint 5	[{39:20%}, {35:20%}, {37:20%}, {31:10%}, {34:20%}, {36:10%}]
ProgramPoint 6	[{50:50%}, {60:50%}]
ProgramPoint 7	[{200: 80%}]

[{observed execution freqeuncy : observed frequency instance}, ...]

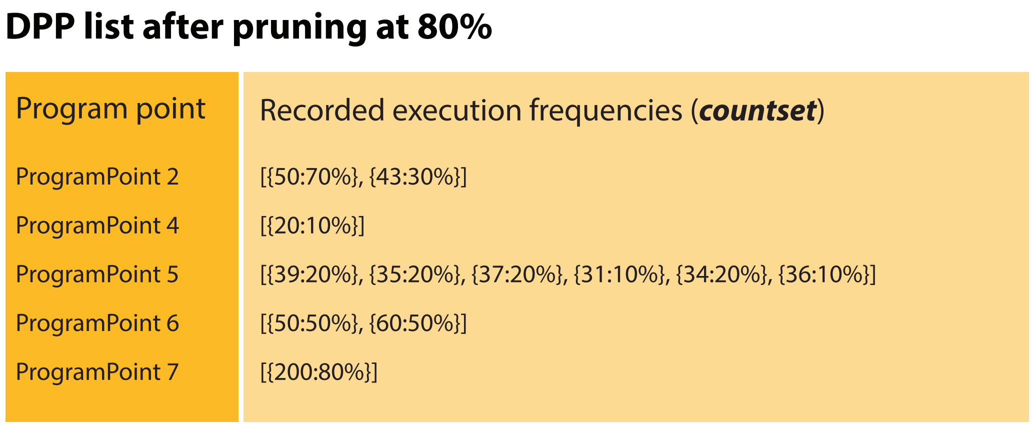
(a) DPPs list with highly deterministic and non-deterministic false positives.



DPP list after pruning at 30%

Program point	Recorded execution frequencies (***countset***)
ProgramPoint 2	[{43:30%}]
ProgramPoint 4	[{20:10%}]

(b) At a pruning threshold of 30% *ProgramPoint7* is missed.



DPP list after pruning at 80%

Program point	Recorded execution frequencies (***countset***)
ProgramPoint 2	[{50:70%}, {43:30%}]
ProgramPoint 4	[{20:10%}]
ProgramPoint 5	[{39:20%}, {35:20%}, {37:20%}, {31:10%}, {34:20%}, {36:10%}]
ProgramPoint 6	[{50:50%}, {60:50%}]
ProgramPoint 7	[{200:80%}]

(c) At a pruning threshold of 80% false positive Program Point 5 & 6 are reported.

Figure 3.3: Pruning of false positives with number of executions (n) = 10.

```
input  : $fset$ - A map of DPPs containing false positives from Algorithm 1
input  : $pThreshold$ - An optional pruning threshold value
output: $pSet$ - A map of DPPs with false positives pruned
1  $pSet \leftarrow fSet$ ;
2  if pThreshold not set then
3      $pThreshold \leftarrow numberOfExecutions * 0.05$;
4  end
5  while fSet has more elements do
6      read next DPP in fSet ;
7      $brVolatility \leftarrow length(DPP.brCountSet)$ ;
8      if brVolatility >= pThreshold then
9          $remove$(pSet.DPP) ;
10         else
11             for countValue in DPP.brCountSet do
12                 if countValue > pThreshold then
13                     $remove$(pSet.DPP.brCountSet[branch][count])
14                 end
15             end
16         end
17     end
18 end
19 return $pSet$;
```

Algorithm 2: Pruning out of false positives.

"non-interesting" to our analysis are eliminated, leaving out only the *"interesting"* ones to be reported. The DPPs that are of interest to our technique include the ones with *rare non-deterministic branch execution counts*. This means those DPPs that are non-deterministic but at the same time their occurrence is rarely observed during program executions. This approach is based on the observation that concurrency bugs that are rare are the hardest to detect and debug, and precisely why our approach focuses on these bugs in the second phase. However, non-rare bugs that have caused a change in the control flow path of the program are also available for inspection in the non-pruned logs. The non-rare bugs may be inspected by turning off pruning altogether or by specifying a higher pruning threshold value. By default, if no pruning threshold value is provided, our approach is to prune everything that's common to 95% of the observed executions (line 3 of the algorithm). Almost all false positives may be eliminated if a pruning threshold value equal to the number of failing program runs (observed in the first phase) is specified. If the pruning threshold value is specified in this manner, then our technique only reports DPPs with *volatility* value equal to the number of failing runs.

Definition 12 (Volatility)**:** *The volatility value for a DPP is dependent on the number of elements in its* countset *compared against the specified pruning threshold value. A DPP is considered highly volatile if its execution countset has a* *higher number of elements than the specified pruning threshold (i.e., its execution count keeps changing every few runs of the program). The execution countset contains the observed execution frequencies for each program point in the program under test.*

Definition 13 (Deterministic DPP)**:** *A deterministic DPP for our approach refers to a program point that has a lower volatility than the specified pruning threshold value. The execution counts for a deterministic DPP are stable, i.e., the program point is executed in every execution of the program.*

Definition 14 (Non-deterministic DPP)**:** *A non-deterministic DPP refers to a program point that has a higher volatility than the specified pruning threshold value. A non-deterministic DPP has different associated execution counts every few executions of the program, i.e., the program point is executed in an arbitrary fashion depending on the execution conditions for different program executions.*

The non-interesting DPPs are those that are either highly deterministic or highly non-deterministic. Our technique eliminates the highly deterministic DPPs (Line 14 of Algorithm 2) because of two reasons: First, highly deterministic bugs should be relatively easier to debug compared to the rare ones, and second, we do not want to overwhelm developers with a long list of DPPs to inspect. The highly non-deterministic DPPs are eliminated (Line 9 of Algorithm 2) because they may introduce false positives in the list of DPPs to be reported. As an example of this type of false positive, consider a producer/consumer example, where the consumer wakes up to check if a condition is satisfied (e.g., the buffer is a certain size) and decides to either continue or go back to sleep based on the outcome of the observed condition. This would introduce false positives since in this case the execution count for conditionals belonging to the consumer threads are highly non-deterministic and may change every few runs of the program. However, since any log pruning performed by our system involves a threshold, we can report the non-rare bugs as well if we are willing to sacrifice accuracy, i.e., accept a higher chance for false positives to appear in the final list of DPPs to be inspected.

The final list of DPPs that are reported to developers for inspection refers to specific variables that belong to conditionals in the program being tested. The approach reports variables that belong to conditionals branches because that is where a program's control flow may diverge. The approach ensures that the recorded DPPs refer to conditionals by instrumenting every conditional branch within the program under test. These reported conditionals must be inspected to arrive at a conclusion regarding their truth. The size of the list of reported DPPs depends on the pruning threshold specified during the second phase or our approach. The pruning threshold may be adjusted as desired. A low pruning threshold value means a slightly higher number of DPPs to inspect and consequently, a higher chance for false positives to sneak into the reported list, whereas a higher pruning threshold value is suitable for bugs that appear rarely during program execution. The variables that are reported might not be involved in the bug by themselves, but are affected by the bug and definitely involved in

Thread 1

```
...
if(status == ‘‘r’’):
        do foo();
else:
        do bar();
...
```

Thread 2

```
...
status = ‘‘r’’;
...
status = ‘‘w’’;
...
```

Figure 3.4: Type 1 - An example of a data race.

Thread 1

```
...
lock(m1)
        fill_up(cache);
unlock(m1)
...
lock(m2)
        if(isEmpty):
                isEmpty = False;
unlock(m2)
...
```

Thread 2

```
...
lock(m1)
        consume(cache);
unlock(m1)
...
lock(m2)
        if(!isEmpty):
                isEmpty = True;
unlock(m2)
...
```

Figure 3.5: Type 2 - An example of an atomicity violation.

its manifestation. Therefore, they provide a good starting point to figure out what could cause that particular conditional or sub-conditional to be either flipped or be executed a certain(different) number of times in a different execution of the program.

3.5 Implementation

We have implemented a prototype of our approach in C++ to evaluate the effectiveness, accuracy, and scalability of our technique. We employ Pin [77] to perform dynamic binary instrumentation of the program under test to collect the conditional branches that are executed at run-time. Every conditional branch that is recorded also has an associated counter with it, which is used to record the number of times each conditional branch is taken during program execution. We consolidate the execution counts for all threads to account for load balancing (i.e., the sum total of counters for all the threads is considered and not that of an individual thread). Binary instrumentation provides a reliable low-cost way to monitor the execution of conditional branches. This information is mapped back to the source language program to allow the user to understand the detected problem in source language terms. The focus of our implementation is mainly to detect and localise concurrency bugs, not trigger them. However, to influence the scheduler and increase the chance of a rare thread interleaving to occur our system may delay random shared memory accesses for a small amount of time. The accesses to be

Thread 1

```
...
lock(m)
        process = True;
        while(process):
                w = True;
                wait(cond,m);
                w = False;
unlock(m)
...
```

Thread 2

```
...
lock(m)
        process = False;
        if(w):
                signal(cond);
unlock(m)
...
```

Figure 3.6: Type 3 - An ordering violation bug.

Thread 1

```
...
lock(m1)
        var1 = True;
unlock(m1)
...
if(var1 and var2){
        ...
}
...
```

Thread 2

```
...
lock(m2)
        var2 = True;
unlock(m2)
...
if(var1 and var2) {
        ...
}
...
```

Figure 3.7: Type 4 - A data race involving multiple variables.

delayed are chosen in an entirely random fashion. Delays are caused by injecting calls to *sleep* inside the program binary for a small random amount of time (in microseconds) before random memory reads and writes. The Pin tool provides the functionality to determine memory read/writes using generic inspection API. In general our approach complements techniques for triggering bugs and may be combined with these schemes to form an effective bug finding strategy. This coupling of our approach with other bug triggering techniques could ensure that once a bug is triggered, it may be localised precisely to a small part of the program without overwhelming developers with a large number of false positives.

The final set of DPP logs produced by our approach are minimal, capturing only the most relevant information. The logs may be further pruned to remove the effects of any inherent non-determinism in the program by adjusting the pruning threshold value. The information presented in these logs is concise and includes only the function name, the image that this function belongs to, the data location for the disassembled instruction, and a list of attached frequencies for the number of times that this branch is taken along with how often each frequency is observed in the recorded executions. This list of reported DPPs is usually quite small in practice, and we have observed that it is no more than ten DPPs in the experiments we performed. The experimental results of our implementation are presented in the next section.

3.6 Evaluation

Our initial evaluation focuses on a number of C/C++ programs as there are many concurrent applications written in these languages; however, the approach is not language-specific and can be employed with other compilers as well. To evaluate the prototype implementation of our approach we selected a number of constructed test programs written in C/C++. These programs are quite small (around 100 lines of code), however, they serve to illustrate the representative bugs in concurrent programs. A simplified version of the bug in each of the constructed representative programs is shown in Figures 3.4 - 3.7. Apart from these constructed examples, we also evaluated against a bug in the data compression utility PBZIP2, an atomicity violation in the multi-threaded download accelerator Aget, as well as a bug in the Apache httpd web server, and an implementation of StringBuffer that mimics its buggy behaviour from jdk 1.4. The aforementioned applications have been constantly utilised in literature for the purpose of evaluating software systems, however, we gathered their details from Yu et. al, and Flanagan et. al [40, 142]. The experiments performed with these applications serve to illustrate the effectiveness, accuracy and scalability of our technique.

```
public synchronized StringBuffer append((char *str) {
        ...
        int len = sb.length();
        ...
        //Another thread may remove characters from sb before the next
        //statement executes, and render the current len value stale.
        //The next call to getChars(...) will throw an outOfBoundsException
        ...
        sb.getChars(0, len, value, count);
        ...
}

public synchronized int length() {
        return count;
}

public synchronized void getChars(...) {
        if (srcBegin < 0) {
                throw exception;
        }
        if ((srcEnd < 0) || (srcEnd > count)) {
                throw exception;
        }
        if (srcBegin > srcEnd) {
                throw exception;
        }
        System.arraycopy(...);
}
```

Figure 3.8: A concurrency bug in stringBufer from jdk 1.4.

In our experiments with these benchmark programs, we succeed to detect these bugs even with a single failed program run with a high level of accuracy in most cases. These benchmark applications are listed in Table 3.1. The first column lists the application names. The second column show the category for each bug according to the class of concurrency bugs that it belongs to. The third column shows the effect that each bug has on the application when the bug is triggered during an execution. The last column provides a short description of each bug. The experimental results for the benchmarks programs are presented in Table 3.2.

Table 3.1: The different types of bugs along with description of each

Application	Bug Type	Bug Effect	Bug Description
Type1	Data Race	May report wrong result	A data race on the variable *status*. Depending on the order of operations between Thread1 and Thread2, either the if() or else part of the statements in Thread1 may be executed, leading to its buggy behaviour.
Type2	Atomicity Violation	Data corruption / loss of data	The program expects the two different locks within the same thread to be acquired without any interruption from other threads. A violation of this assumption causes the bug.
Type3	Ordering Violation	Indefinite wait	An example of an ordering violation leading to a lost wake up signal. If the statements in thread2 are executed before those in Thread1, then Thread1 waits forever for a signal that never arrives.
Type4	Data Race, Multi-variable	May report wrong result	A data race involving multiple variables, where each of the individual variable is accessed using different locks, but later both of the variables are used together to perform some operation.
pbzip2	Order Violation	Segmentation Fault / Crash	An order violation between the main thread and the consumer threads. The consumer threads are expected to terminate before the main thread releases the mutex *fifo->mut*. However, when this order is violated, the application crashes with a segmentation fault.

Continued on next page

Table 3.1 – *Continued from previous page*

Application	Bug Type	Bug Effect	Bug Description
StringBuffer	Atomicity Violation	Array out of Bounds Exception	An atomicity violation (shown in Figure 3.8). Triggered when *append()* on Line 3 calls the *synchronized* method *length()* on some buffer *sb*, another thread could remove characters from this buffer, making the previously calculated length of the buffer stale. This culminates in a call to *getChars()* method on Line 9 with invalid arguments (i.e., wrong length) and results in the bug.
Apache	Data Race	Garbage in error logs	The data race surfaces after a graceful restart is requested by user with Apache httpd 2.2.15. The child thread closes all its listeners (with dummy connection), returns from *pr_pollset_poll* with the socket and calls *lr->accept_func()*. However, the child process has already set *die_now* and has closed the listeners. The server logs an error, reporting a bad file descriptor.
Aget	Atomicity Violation	Inconsistent logs and download	A mix of data race and atomicity violation on the variable *bwritten*. The bug occurs when user cancels the download by issuing *ctrl+c* from the console. The signal handler then attempts to save progress by calling *save_log()*, where the variable *bwritten* is accessed without holding the mutex lock. In such a scenario, it is likely that the downloaded file and the log file are inconsistent.

3.6.1 Accuracy

To evaluate the accuracy of our approach, we executed each of the benchmark programs listed in Table 3.2 *n=100* times under the control of our tool. In the context of our work, we define accuracy as under:

Definition 15 (Accuracy)**:** *The accuracy for our system refers to the ratio of true*

Table 3.2: The pruned list of DPPs reported for all failed runs.

Application	Distinct branches	Full DPP	Pruned
Type1	9	4	2
Type2	8	2	0
Type3	8	5	2
Type4	11	2	1
pbzip2	422	5	1
Aget	172	6	5
stringBuffer	245	12	6
Apache httpd	344	6	6

bugs as compared to the reported false positives i.e., a lower false positive rate when compiling the list of DPPs that are involved in the manifestation of a bug.

For the experiments performed as part of the evaluation, a value of 100 is chosen for n. However, choosing a lower or higher value for n would produce similar results, since our pruning algorithm would prune out the non-interesting events. For a small value of n the pruning threshold may need to be adjusted accordingly to a higher value (as demonstrated by the example in Figure 3.3, where the pruning threshold is set to 30% for the n value of 10). The value for n has no special relationship with accuracy in this case. We use n=100 to show the performance of our offline analysis. In practice one may stop as soon as a failing execution of the program is observed and at least one or more successful executions are logged (or vice versa). This observation is confirmed by our experiments with Aget, where a value of 5 was chosen for n. In each of the listed cases, only a single failing execution profile of the application under test is used. The use of a single failing execution profile confirms that our tool is able to localise bugs accurately without requiring multiple failed executions profiles, as is the case with most current approaches in literature. Table 3.2 shows the results of our experiments with the aforementioned parameters. The first column lists the types of applications that were tested, the second column gives a rough idea of the scale of these applications by listing the number of branches that are executed for a single input. The third column lists the DPPs that are reported in the full log files[5], while the last column shows the number of DPPs that are reported after executing our pruning technique to eliminate the highly non-deterministic DPPs.

As we can see from Table 3.2, for StringBuffer, when it is executed 100 times with no failing runs the total reported DPPs that are executed by the application are around 245. Out of the total collected DPPs, six are executed in a highly non-deterministic fashion. When the same 100 executions were analysed with just a single failing run added to the mix, the number of DPPs reported comes out to be 12 (out of which six non-deterministic DPPs come from successful executions). When these logs are pruned with the 95% default pruning threshold value, the six highly

[5]This list excludes the highly deterministic DPPs. The execution counts for these deterministic DPPs never change in the observed execution profiles.

non-deterministic DPPs are not reported any more. The DPPs that are reported are the ones that are involved in the manifestation of the bug. These reported DPPs are otherwise highly deterministic on successful executions of the program, i.e., their count is constant for 100 executions but only different for a single program run. Further inspection reveals that out of these six DPPs that are reported, one belongs to the test harness that is used to trigger the bug, while two of these belong to the append() method, which calculates the length for stringbuffer. The remaining three DPPs belong to conditionals performing bounds checks for the getChars() method, which — when called with a wrong value for length — causes the bug and throws an exception. The case for bugs that follow the pattern of Type2 is slightly special as shown in the example of Figure 3.3 in Section 3.4.2. In the case for bugs of Type2 the compiler reverses the conditional, so that *not* executing that branch is buggy behaviour. This bug only appears in the list of reported DPPs if we increase the pruning threshold to 99% (i.e., we only eliminate the most deterministic branches and report everything that is non-deterministic in the slightest). This applies to our case, since we have a single failing program run, so with our default pruning threshold, this DPP is pruned out and is not reported to the user. This DPP is reported if an extremely high pruning threshold value is specified, or no pruning is performed and the full list of two DPPs is inspected.

Similarly for pbzip2, the total number of branches that are executed for a particular input comes out to be around 400. The report consists of five DPPs out of which only four are non-deterministic. These non-deterministic DPPs are a result of the producer/consumer model that the application follows. More specifically, pbzip launches producer threads that read in blocks of data for compression, and puts them on a queue for the consumer threads to later read and perform the compression. The consumer threads perform a *conditional timed wait* for producer threads to put data on the queue and notify them. The conditional timed wait implies that consumer threads must wait for a certain amount of time (one second in this case) for the producer threads to signal that more data are available for processing. If the consumer threads do not receive a signal within the specified time, they will go ahead and check the queue anyway to see if there is some data available for processing. This non-deterministic behaviour of the consumer threads causes certain conditional checks to be performed non-deterministically, and depending on the outcome of the condition either compression is performed, or the consumer threads go back to a state of conditional waiting. Checking for the bug with our approach, while using just a single failing run, we are able to determine the location in the program where it deviates from its expected behaviour and to report only a single DPP with our default pruning threshold. This DPP correctly points to the *if* condition on Line 5 of Figure 3.1, which is the branch where this bug actually manifests itself.

The bug in Apache httpd and Aget are slightly different compared to the rest of the bugs that we used for the experimental evaluation of our approach. The bugs in Apache httpd and Aget do not change the control flow path of the program, and therefore, our approach is not able to detect these bugs.

In the case of Aget the five extra DPPs that are reported are a result of the non-deterministic nature of the application. The results for Aget are based on five executions of the program that includes one failing run. A small number of executions are recorded in the case for Aget to show that our technique does not require a large number of recorded program executions to work. The false positive DPPs that are reported for Aget are successfully pruned out when further successful program runs are collected. The false positive DPPs belong to code regions that deal with updating the progress bar and depend on the buffer size as well as the download speed. The aforementioned dependencies render the execution of the false positive DPPs highly non-deterministic.

The constructed examples that we present are small programs compared to the rest but serve to illustrate that our approach can detect every type of concurrency bug that causes any change to control flow path of the program. Furthermore, the experiments confirm that our technique is not affected by the size of applications, as we have clearly shown with Apache and pbzip. The approach scales to larger applications like Apache httpd web server without requiring any further effort and is able to detect any of the patterns described in Figures 3.4 - 3.7.

3.6.2 Scalability

The applications selected to evaluate the scalability of our approach are of varying sizes. The size of selected applications range from roughly two thousand lines of code to two hundred thousand lines of code in the case of Apache. The constructed programs and the stringBuffer from Java's JDK are between 100 and 200 lines of code large, and are therefore not included in the experiments for evaluating the performance of our technique.

Our results with Apache confirm our claim that our system is not affected by the scale of applications being tested. For smaller programs like pbzip our approach adds more overhead compared to Apache because apart from the constant time overhead of loading everything into Pin, the size of pbzip compared to Apache is quite small. Furthermore, for pbzip the number of conditional branches that are executed and consequently instrumented is roughly around 400, whereas for Apache the number of these branches is roughly 1000 (including those for wget). If we compare these numbers to the size of the application in each case, we can see that our technique is not affected by how large an application is (i.e., two thousand vs two hundred thousand lines of code).

3.6.3 Performance

To evaluate the performance of our online profiling tool, we show the runtime overhead incurred by the profiler in Table 3.3. These experiments are performed on a desktop machine with an Intel core i7 3.50GHz processor and 16GB of memory, running Ubuntu desktop 14.04 operating system. In all of the cases our technique performs much better than even the most recent bug localisation approaches. All the experiments are repeated 500 times to make sure that the data are reproducible. In the experiments for pbzip, a test input file is provided to be compressed. In the case for Apache web server, the server is first started, then an image file is downloaded from the server. After the download is completed, the file is deleted and the server is restarted. The downloaded file is a 43MB Jpeg image file. In the case of Aget the experiment includes downloading the same 43MB Jpeg file from our local server and removing the file before repeating the experiment.

Table 3.3: Normalised overhead of our offline and online analysis compared against other approaches.

Application	Offline Analysis	Online Profiling	LOCON	Falcon
pbzip2	0.065 seconds	17.12x	19.96x	17.85x
Apache	0.091 seconds	3.97x	43.75x	44.25x
Aget	0.067 seconds	2.75x	41.34x	49.52x

The results of these experiments are listed in Table 3.3. In each case we can see that our tool clearly outperforms both Falcon[6] [101] and LOCON [132]. In the case of pbzip, Falcon and LOCON report an overhead of 17.85$\times$ and 19.96$\times$ respectively, whereas the overhead with our system is 17.12$\times$. In the other two instances, with both Apache and Aget we are between 11$\times$ to 18$\times$ faster than LOCON and Falcon. The reason why we are almost 11$\times$ faster is that both of these tools need to instrument and collect information at every shared memory access, and therefore the overhead is proportional to the size of the program being tested. In our case our tool only instruments and collects data at every conditional branch that is entered, and therefore our approach incurs a lower overhead.

The offline analysis of our approach that is performed in the second phase depends on the number of executions recorded in the first phase. We present these numbers in the first column of Table 3.3. We select 100 log files to analyse as this is the same number used for localising bugs in majority of the cases in Table 3.2. The overhead includes pruning of logs with the default pruning threshold of 95%. Our experiments show that this overhead is quite low. Even when the number of log files to be analysed is pushed up to 500, the overhead comes to around 0.22 seconds in the case of pbzip. A similar story is repeated in the case of Apache when the number of executions to analyse is increased to 500, the total time spent is 0.31 seconds.

3.6.4 Threats to validity

Concurrency bugs such as data races, atomicity violations or ordering bugs may or may not affect the control flow of a program. If a bug does not change the control flow of a program, then our approach may miss this bug. In such a situation where the control flow remains unchanged, programmers may be lured to false positives. However, if the number of failing program runs (or successful

[6]We use the performance numbers mentioned in [132] for their C++ implementation of Falcon.

ones) is known beforehand, then this trap may be avoided by specifying a pruning threshold accordingly. That is, specifying a pruning threshold equal to the number of observed failing program runs. Further, since our log pruning procedure uses a threshold and we prune logs aggressively to avoid reporting false positives, our technique might end up pruning a DPP that has a low volatility but still points to a bug. This can happen in cases where the bug manifests itself when the program does not execute a conditional branch, i.e., the branch is always executed on successful runs but never on failed runs. This means that if we have only a single failed run, then this lowers the volatility for the branch that is not executed, and it may results in that branch getting pruned, even though it is pointing to a bug. To avoid this situation, we have two options: either we can collect more failing runs, which may be quite difficult in practice, or we can use a lower pruning threshold during the second phase of our approach. The second option of lowering the pruning threshold works out quite well in practice. For example, even if we prune the highly deterministic DPPs (requiring an extremely low pruning threshold), leaving only the list of non-deterministic DPPs to be examined, this list is still quite small. In the case for stringbuffer this list shows only 12 DPPs, six of which are false positives, and only five DPPs in the case for pbzip2, out of which four are false positives. Even in the case of a large application like apache, only five false positive DPPs are reported. The low false positive rate makes it feasible to confirm whether a DPP points to a bug or is merely a false positive.

3.7 Concluding remarks

We have presented a technique for detecting and localising concurrency bugs and an evaluation using a prototype implementation. The technique can detect and localise concurrency bugs without any discrimination of the type of bug involved. In most cases this technique has a high level of accuracy, i.e., it points the user to program code where the bug has an impact on the execution profile. We have shown that these bugs can be localised with only a single buggy execution. Our approach can guide the user to a point in the program's source code that reflects where the behaviour deviates from that of a successful program execution. The technique is scalable, as we have shown by localising a bug in pbzip2 as well with our experiments of Apache. Although our initial focus has been C/C++ applications, this approach is language-agnostic and can be applied to other languages as well.

The technique is not influenced by the size of the applications or the number of threads. Furthermore, our system can cope with the effects of load balancing and the inherent non-determinism in applications, since these two aspects lead to false positives. We have shown that we are able to successfully prune all false positives in most cases. However, there are cases where false positives may be reported as a consequence of specifying a higher pruning threshold. The number of these reported false positives is still quite small and may require little effort to confirm.

The approach employs binary rewriting to collect data about a program's execution. Binary instrumentation, although it incurs an overhead, nevertheless results in better performance than other approaches and allows the use of this technique in realistic settings. The overhead is probably too high to allow the system to be used by default, but the low overhead compared to other approaches (and the fact that a single buggy execution is needed) allows its use as soon as a problem is suspected.

The decision to use conditional branch counts in the prototype implementation is based on the fact that current support for monitoring the execution behaviour of programs is limited, especially for multi-threaded programs. Other options (e.g., to replace binary instrumentation by a compiler framework to keep track of the conditional branches that are executed and to directly use the processor's program monitoring unit) are possible and worthy of further investigation. However, the difficulties to obtain valuable data about a program's execution points to the need for better program monitoring units. Given the proliferation of multicore processors, and the ability to provide additional functionality in processor implementations, architects of future processors should pay more attention to the needs of tools that help software engineers isolate and understand program defects.

4 Profiling for detecting performance bugs

This chapter introduces a profiling technique for detecting performance bugs in concurrent software. The approach can identify performance bottlenecks in real-world software using a hybrid of dynamic and post-mortem analysis techniques.

4.1 Overview

Identifying and understanding performance problems is difficult for concurrent software but is an essential part of the software development process for parallel systems. In addition to the same problems that exist when analysing sequential programs, software development tools for parallel systems must handle the large number of execution engines (cores) that result in different (possibly non-deterministic) schedules for different executions. Understanding where exactly a concurrent program spends its time (especially if some aspects of the program paths depend on input data) is the first step towards improving program quality. Most state-of-the-art profilers, however, aid developers in performance diagnosis by providing hotness information at the level of a class or method (function) and usually report data for a single program execution [145].

The approach introduces a profiling and analysis technique that consolidates execution information for multiple program executions. Currently, our tool's focus is on execution time (CPU cycles), but other metrics (stall cycles for functional units, cache miss rates) are possible, provided such data can be obtained from the processor's monitoring unit. To detect the location of performance anomalies that are worth addressing, we take into account the average amount of time spent inside a code block along with the statistical range of the minimum and maximum amount of time spent.

The technique identifies performance bottlenecks at the fine-grained level of a basic block. It can indicate the probability of such a performance bottleneck appearing during actual program executions. The technique utilises profiling information across a range of inputs and tries to induce performance bottlenecks by delaying random memory accesses. The approach is evaluated by performing experiments on the data compression tool pbzip2, the multi-threaded download accelerator axel, the open source security scanner Nmap and the Apache httpd web server. An experimental evaluation shows the tool to be effective. Modifications in the block that is identified by the tool result in performance improvement of over 2.6$\times$ in one case, compared to the original version of the program. The performance overhead incurred by the tool is a reasonable 2-7$\times$ in majority of cases, however, in the case of pbzip it goes up to 28$\times$.

4.2 Background

Performance problems are becoming more prevalent in modern software systems [56, 62, 75, 85, 127] as software grows in size. The widespread adoption of multi-core processors and the use of concurrent and multi-threaded programming to utilise these machines have contributed heavily to performance problems. Software correctness and safety constraints in the presence of concurrency may exacerbate the situation even further and lead to performance anomalies in released software. These performance anomalies render software slow and unresponsive. Performance problems may lead to poor user experience, lower system throughput, and a waste of computational resources [14]. Software that is slow may not only frustrate end users but also cause financial losses [56]. Following the convention adopted by other researchers in dealing with performance issues [56, 95, 124], we consider an anomaly to be a performance bug if it affects the speed or responsiveness of a program and relatively simple source code changes may lead to significant improvement in performance while preserving functionality.[1]

Performance anomalies exist widely in production software. For example, Jin et al. [56] report that Mozilla developers have fixed 5-60 performance bugs reported by users every month over the past 10 years. There are many sources of inefficiency (e.g., poor design or inefficient data structures), and these problems may plague sequential programs, as well. Concurrent software is subject to the same issues, but a software developer must in addition deal with unexpected or rare resource contention scenarios involving multiple program threads. The rare resource contention scenarios cause performance anomalies in programs and may only manifest under rare circumstances, such as a unique interleaving of the program threads. Furthermore, safety/correctness and performance are two goals for software developers that sometimes exert contradicting pressures. The goal of correctness requires programs to be thread-safe — and as a consequence may require additional synchronization; the goal of fast execution may encourage developers to minimize synchronization in a program.

Performance bugs have only rarely been the focus of research on software

[1]Our goal is to identify possible performance anomalies but possible transformations to address the performance issue are outside of the scope of this chapter. We expect a user to analyse the situation and to modify the source code, or to employ a different data structure, or to use a different algorithm.

defects. There are a number of reasons why performance bugs have received considerably less attention than their functional counterparts. First, developers can check for functional correctness by relying on well established testing techniques for detecting functional bugs [16, 37, 60, 132]. Second, there exist formal approaches to software verification and testing based on symbolic execution, which allow (some degree of) automation [19, 20]. Third, most approaches for performance modelling of parallel and distributed systems offer limited support for performance oriented software engineering, or they are capable of dealing with only small programs [21, 106]. Lastly, performance bugs are hard to diagnose because of their non-failure semantics, i.e., they do not cause failures or program crashes. Furthermore, existing approaches for detecting software performance problems are limited in their functionality and report on resource consumption for a single program execution. The resource consumption of a program is usually dependent on the efficiency of the algorithm being used and the size of the program input. Concurrent programs further complicate matters because of their non-deterministic nature and large number of possible thread schedules. Different program inputs or thread schedules may lead to different program behaviour. The state-of-art for performance bug diagnosis is the use of profiling tools (e.g., oprof or gprof [46, 71]). However, profiling tools are generally limited in their functionality. CPU time profilers usually report performance bugs at the level of hot methods/routines for a single program execution. These reported methods can be arbitrarily long. Locating the source of these computationally intensive and time-consuming code regions inside a method can prove to be a challenging task in large software systems.

Profiling tools often fail to discover dormant performance problems that may appear under rare circumstances. Similarly, compilers might miss optimising a routine that has (real or assumed) side-effects, and identifying the places where cycles are wasted is far from easy. One of the key reason for the lack of better profiling and detection tools for performance bugs is that developers focus on removing functional bugs, and because better tool support is available for functional bugs. In contrast to procedures and tools for detecting functional bugs, tools and procedures for performance bugs lack functionality.

4.2.1 Contributions

Although current software tools for profiling performance anomalies are useful, they still lack functionality. One of the major problems with software profiling techniques is that they provide information only about a specific program run. However, in realistic settings, it is quite likely that different executions of a program with different inputs and/or different thread schedules may lead to different execution profiles of the program. (The program still computes the intended result — otherwise there is a functional bug as well.) The approach presented in this chapter detects performance bottlenecks by dynamically collecting profiling information about program executions. The approach aggregates information from multiple executions to detect non-deterministic or rare performance bottlenecks. Rare bottlenecks require some specific program input and/or environment conditions to manifest. Performance bottlenecks that are deterministic are detected and localised with a single profiling run of the program.

In short, the approach presented in this chapter makes the following key contributions.

- We present an approach that combines multiple execution profiles of the program to provide an overall picture of the resource consumption for a wide range of inputs. The use of multiple execution profiles enables our approach to report on rare resource contention scenarios. Our profiler does not rely on hotness information for a single input alone. Instead, the profiler uses information about the amount of CPU execution time taken by a basic block for a range of inputs during different executions.

- The approach provides information about performance bugs at a finer-grain level than earlier tools by identifying the relevant basic block(s) instead of reporting an entire function or class. In certain cases modifying the identified basic block leads to an improvement in program performance of more than 2.6× (details in Section 4.5). The approach is not prone to report false positives since it reports those basic blocks whose contribution to the execution time exceeds the given threshold (i.e., percentage of the program's total execution time). Our approach eliminates false positives that always take a large amount of time by using the statistical range of the CPU execution time spent inside a basic block.

4.3 Approach and methodology

The approach for obtaining basic blocks that contribute most to the overall execution time of a program is divided into two phases. These two phases are summarised in Algorithm 3 and Algorithm 4.

The first phase of our approach is concerned with on-line profiling of the program. In this phase, the program under test is executed multiple times with different inputs.[2] For each execution of the program, our tool records hotness information and the amount of CPU execution time taken by every basic block. In the context of our tool, hotness of a basic block refers to how often (i.e., frequently) a basic block is executed during a given program execution. The execution frequency of a basic block contributes to its hotness, i.e., the more frequently a basic block is executed, the hotter it becomes. The tool records and maintains the minimum, maximum, and average amount of CPU execution time for each basic block in the program under test. The average time is calculated by maintaining a moving average of the CPU execution time, which makes it possible to capture the

[2]Inputs can be supplied by the user or a separate program to generate test input. See the extensive literature on test generation, but this topic is beyond the scope of this chapter.

effects of non-deterministic events (e.g., a call to *sleep()* for some random amount of time) and to eliminate outliers. The diversity of inputs to the program as well as the number of times the program is to be executed is for the user to decide. In this phase the tool tries to influence the scheduler to cause different schedules of the executing program threads. Influencing the scheduler makes it possible for the tool to unearth rare resource contention scenarios that may otherwise go unnoticed during normal testing. Influencing the scheduler is achieved by delaying random shared memory accesses for a small amount of time. At the end of this phase, the profiled information for each execution is written to disk and is available for analysis in the second phase of our approach.

```
input  : $PUT$ - Program under test
output: $profile$ - A log file containing hotness and timing information per
        basic block
instrument(PUT) ;
while PUT.isAlive do
    for each BB in PUT do
        execTime=BB.exitTime-BB.entryTime;
        if BB.execFreq $>$1 then
            BB.avgTime= (execTime +
              (BB.avgTime*BB.execFreq))/(BB.execFreq+1);
            BB.minTime = min(BB.minTime, execTime);
            BB.maxTime = max(BB.maxTime, execTime);
        end
        else
            BB.avgTime = BB.minTime= BB.maxTime = execTime;
        end
        BB.execFreq+=1;
    end
end
WriteProfileLog(PUT);
```

Algorithm 3: Phase 1: On-line profiling of the programs under test

The second phase of our approach is outlined in Algorithm 4. This phase performs off-line analysis of the data recorded during the first phase. In this phase, the recorded information is analysed to identify potential *performance bottlenecks*. The bottlenecks correspond to basic blocks inside the program under test. Performance bottlenecks for our approach are those basic basic blocks where the program spends a *large*[3] portion of its overall execution time. The overall execution time is the total CPU execution time taken by the entire program in a single execution. If the portion of execution time spent in a basic block and the statistical range of the execution time spent in that basic block are above a certain threshold (i.e., percentage of the total execution time), then the basic

[3]The programmer decides what is to be considered *large* by specifying a threshold.

```
input  : profileLog - A log of profiled data from Algorithm 3
input  : eThreshold - A threshold percentage value of total program execution
         time
output: pbList - A list of potential performance bottlenecks
1  totalExecTime = readProfile(profileLog) ;
2  eThreshold ← (totalExecTime*eThreshold)/100;
3  bbList ← empty ;
4  while more profileLog exist do
5      read(profileLog);
6      for each BB in profileLog do
7          if bbList.exists(BB) then
8              prevAvg = bbList.BB.avgTime;
9              prevFreq = bbList.BB.execFreq;
10             prevMin = bbList.BB.minTime;
11             prevMax = bbList.BB.maxTime;
12             bbList.BB.avgTime = max(BB.AvgTime, prevAvg);
13             bbList.BB.execFreq = max(BB.execFreq, prevFreq);
14             bbList.BB.minTime = min(BB.minTime, prevMin);
15             bbList.BB.maxTime = max(BB.maxTime, prevMax);
16         end
17         bbList.add(BB);
18     end
19 end
20 for each BB in bbList do
21     bbSig ← BB.execFreq * BB.avgTime;
22     bbRange ← BB.maxTime - BB.MinTime;
23     if bbSig >= eThreshold and bbRange >= eThreshold then
24         pbList.add(BB);
25     end
26 end
27 Report(pbList);
```

Algorithm 4: Phase 2: Off-line analysis and reporting of performance bottlenecks

block is reported as a potential performance bottleneck. The threshold value of the total execution time may be optionally specified by the user, to override the default value of five percent that our tool uses. Any basic block for which the total execution time and the statistical range of execution times is equal to or above the threshold specified by the user is reported for further inspection. For a basic block to be reported as a performance bottleneck, its significance value as well as range must be higher than the specified threshold. The significance value is a product of the observed average time and the *execution frequency* for a basic block in the recorded program executions (line 20 of Algorithm 4). The statistical range is the difference between the lowest and highest amount of CPU execution time observed in the recorded executions (line 22 of Algorithm 4). Execution frequency refers the number of times a basic block is executed during a single program run.

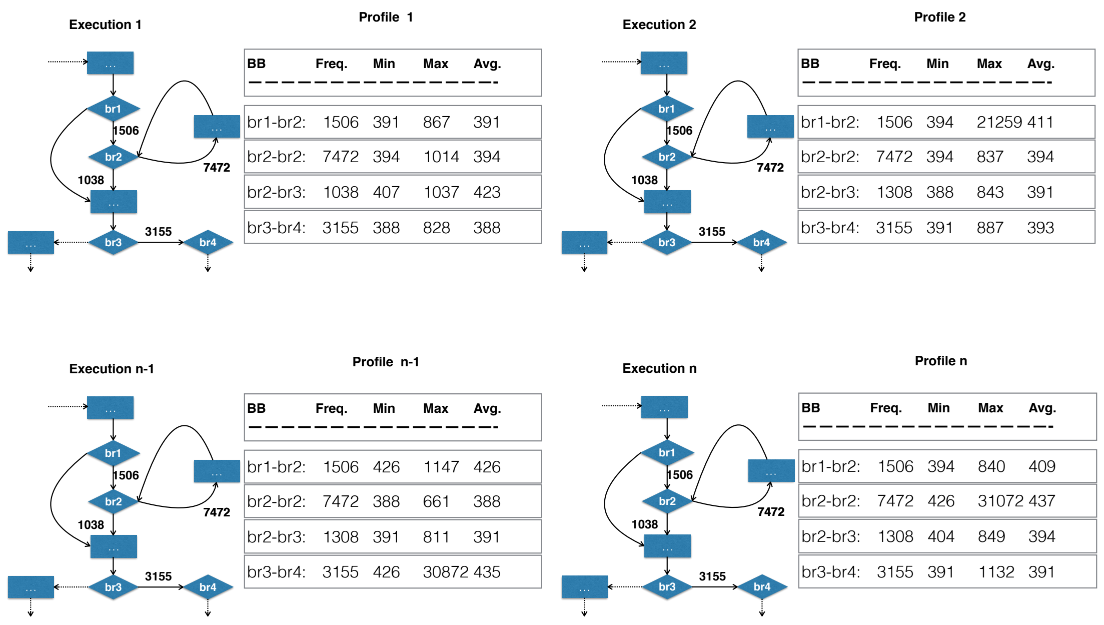
To perform the pruning of false positives and to identify potential performance bottlenecks, our approach chooses the highest recorded value for the execution frequency, the average and maximum amount of execution time, and the lowest recorded value for minimum execution time when aggregating these values from multiple profiled program executions. Selecting recorded values in the aforementioned fashion may be important because non-deterministic execution scenarios involving contention for resources leading to performance bugs may be of more concern to programmers than just average values.

The final step in the second phase of our approach reports potential performance bottlenecks to programmers. The performance bottlenecks report consists of (for each potential bottleneck) the recorded execution frequencies of the basic block in the profiled program runs, as well as the average, minimum, and maximum amount of CPU execution time recorded for each basic block. The execution frequencies for a basic block signify its hotness across different program inputs as well as whether the basic block is executed in a deterministic/non-deterministic fashion in the recorded program executions. The minimum and maximum CPU execution times show the range of time spent in a basic block for different program inputs, while the average amount of CPU times show the average time spent in a basic block in all the recorded execution of the program. The reported values may be further analysed by using a box plot for the reported values, or the standard deviation for the values may be computed before further investigation.

4.4 Implementation

The prototype implementation of our approach uses dynamic binary instrumentation using the Pin [77] instrumentation framework to collect run-time information about the execution behaviour of the program. At run-time, the tool keeps track of the hotness for each basic block by keeping track of how often the basic block is executed during a single program run. Basic blocks are delineated by conditional branches taken by the program (i.e., executing a conditional jump in the program creates a new basic block). Figure 4.1 shows the process of collecting dynamic execution profiles of the program under test. For each basic block our tool stores the minimum and maximum amount of CPU time consumed by each basic block, and computes and stores a moving average of the time spent in the basic block during a single execution. For each execution of the program with the same or different input, this information is stored permanently and written to a file on disk. Storing information in a permanent location (disk) ensures that profiling information is not lost, since our profiling procedure takes a large portion of the total analysis time. At the end of the profiling phase, the log files containing the recorded information from several executions of the program are available for the analysis phase.

During the analysis phase the tool reads the information from log files containing the profiling information and feeds it into the analysis engine. During this phase



Profile 1

BB	Freq.	Min	Max	Avg.
br1-br2:	1506	391	867	391
br2-br2:	7472	394	1014	394
br2-br3:	1038	407	1037	423
br3-br4:	3155	388	828	388

Profile 2

BB	Freq.	Min	Max	Avg.
br1-br2:	1506	394	21259	411
br2-br2:	7472	394	837	394
br2-br3:	1308	388	843	391
br3-br4:	3155	391	887	393

Profile n-1

BB	Freq.	Min	Max	Avg.
br1-br2:	1506	426	1147	426
br2-br2:	7472	388	661	388
br2-br3:	1308	391	811	391
br3-br4:	3155	426	30872	435

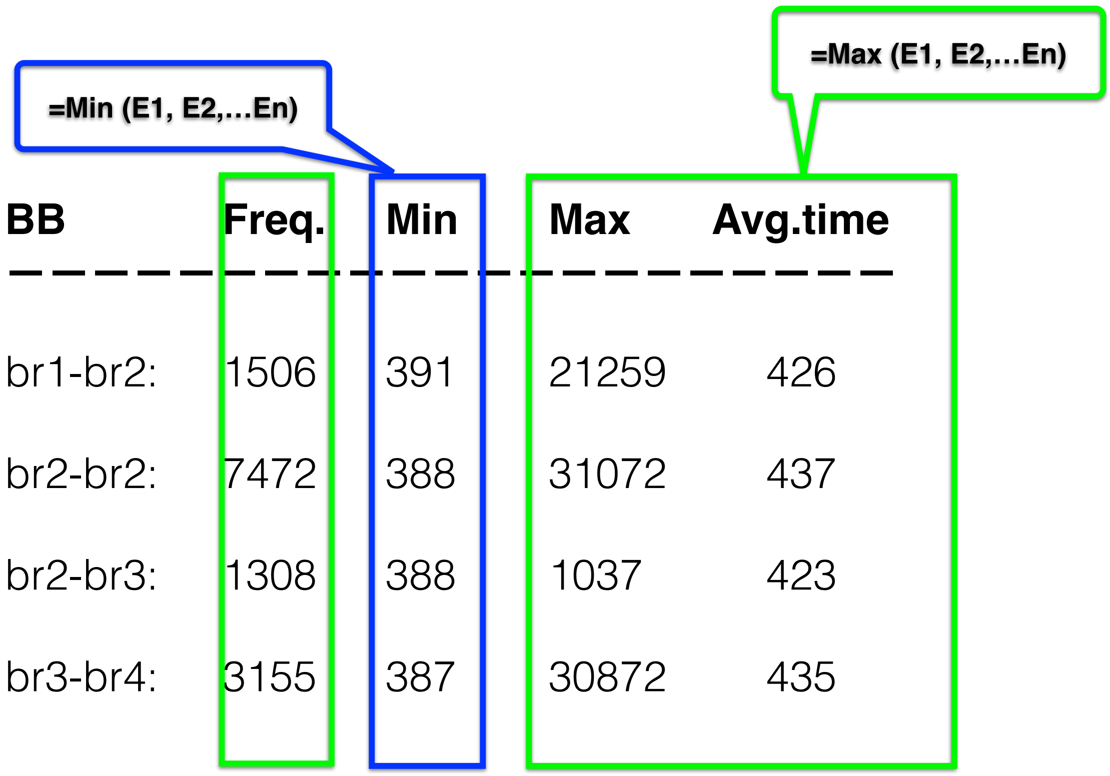
Profile n

BB	Freq.	Min	Max	Avg.
br1-br2:	1506	394	840	409
br2-br2:	7472	426	31072	437
br2-br3:	1308	404	849	394
br3-br4:	3155	391	1132	391

Figure 4.1: Samples of captured dynamic program profiles (Profiles 1 and 2 as well as (*n-1*) and *n* are shown)

occurrence probabilities are calculated for each basic block. The occurrence probability for a basic block indicates the likelihood for a basic block to be executed during actual program runs. The analysis keeps track of the overall minimum and maximum amount of CPU time, as well as the maximum value for the frequency and the average in all of the recorded executions. Figure 4.2 graphically depicts the process of selecting these aforementioned values that are later used to perform pruning of basic blocks that are below the threshold specified by the user.

The analysis phase makes use of a threshold value to judge whether a basic block is a potential performance bottleneck. The threshold corresponds to a percentage portion of the total execution time of the program. For example, the programmer might be concerned about a basic block that takes more than a certain portion of the execution time. For each basic block, the tool compares the product of its hotness and average amount of CPU time as well as the statistical range of the CPU time spent in the basic block with the threshold value. If the product and the range are higher than or equal to the threshold value, then the basic block is reported to the programmer. Otherwise, it is removed from the list of reported basic blocks. The format for the report containing potential performance bottlenecks which is presented to the user is shown graphically in Figure 4.3. The analysis utilises the product of maximum average time profiled in all executions of the program, and the maximum frequency for a basic block to ensure that basic blocks consuming a high portion of the program execution time are identified. However, the basic blocks identified using the product of average time and frequency may very well be a consequence of the algorithm being used by the program. The analysis therefore


=Min (E1, E2,...En)
=Max (E1, E2,...En)

BB	Freq.	Min	Max	Avg.time
br1-br2:	1506	391	21259	426
br2-br2:	7472	388	31072	437
br2-br3:	1308	388	1037	423
br3-br4:	3155	387	30872	435

Figure 4.2: Program profile to prune out basic blocks below the threshold.

utilises the statistical range of the time spent in a basic block in conjunction with the aforementioned product value to ensure that non-deterministic events or contention for resources are accounted for and reported.

The reported information is mapped back to the original program, so that it may be understood in source language terms for programmer inspection.

4.5 Evaluation

We applied the tool implementing our approach to real-world applications written in C/C++. The selected applications include the *pbzip2* data compression tool (version 0.9.4) [42], the *Nmap* security scanner for network exploration and audits (version 6.49) [78], and the *axel* download accelerator (version 2.3) [140]. The results are presented in Table 4.1 and 4.2. The data in Table 4.1 show the results for experiments performed with a single execution profile, while data in Table 4.2 show the results for a setup in which multiple profiles of the program under test are recorded and analysed.

4.5.1 pbzip2

The experiments for pbzip2 using a single execution profile reveal that it spends around 5% of its total execution time inside a *for* loop in the block sorting algorithm of the bzip2 utility. Bzip2 is the non-parallel data compression part of the pbzip2

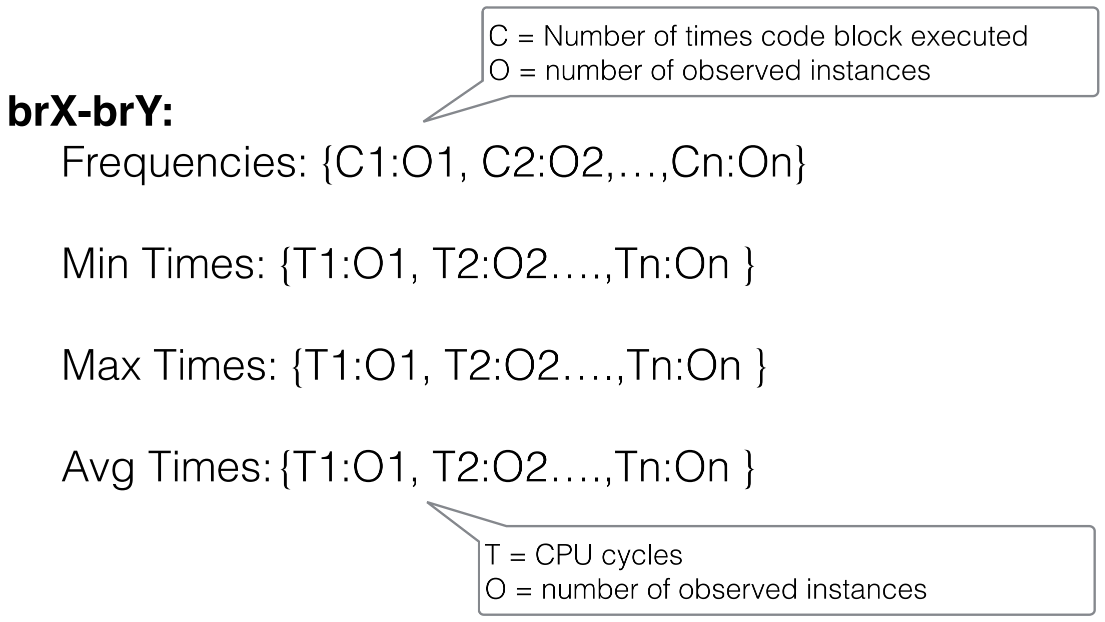

C = Number of times code block executed
O = number of observed instances
brX-brY:
Frequencies: {C1:O1, C2:O2,…,Cn:On}
Min Times: {T1:O1, T2:O2….,Tn:On }
Max Times: {T1:O1, T2:O2….,Tn:On }
Avg Times:{T1:O1, T2:O2….,Tn:On }
T = CPU cycles
O = number of observed instances

Figure 4.3: Format of the report for each identified basic block.

utility. The block sorting algorithm in the application uses Knuth's sequence of increments as part of its sorting algorithm that uses a Shell Sort [122]. The algorithm uses three increment copies. However, the third copy fails to execute most of the time and program execution breaks off back to the main loop, in order to select a new increment. This execution behaviour suggests that the sorting algorithm could be performed without using the code for selecting the third increment. Although this modification may have little impact on improving the performance of the program, it may nevertheless improve code readability and conciseness. We performed the experiments again with the same input, after modifying the bzip2 implementation to remove the third copy that never executed. These experiments revealed a slightly over 0.24% increase in performance with a few inputs. However, this behaviour may need to be investigated further and with more inputs, before we can arrive at any conclusions. To further validate the technique, an artificial workload was introduced inside of pbzip. The experiments were run again to find out if the newly inserted artificial bottleneck is detected, and whether the tool would point to the exact basic block. The artificial workload consisted of a *while* loop involving some basic arithmetic. The workload added slightly over 5% performance overhead to the original version of the program. The threshold execution portion was set to 5% of the total execution time of the program as before. The bottleneck was correctly identified and localised to the inserted *while* loop using logs from just one execution profile. The bottleneck identified in the original version of the program also appeared in the list of reported bottlenecks as expected. Recording and analysing multiple execution profiles for pbzip2 report another 3 bottlenecks that point to the compression process inside of bzip2.

4.5.2 axel

Our experiments with axel[4] reported four potential performance bottlenecks. The bottlenecks belong to a *for* loop and the *if* conditionals inside it, that are used for displaying download progress and speed. Since progress is to be displayed and updated continuously, on a per kilobyte basis, the procedure takes a major chunk of the overall program execution. To validate the experiment, the progress and speed reporting routine was changed from per kilobyte to per megabyte. These minor changes led to a significant improvement in performance. Although we realise that reporting download progress per megabyte might not be suitable for smaller download sizes, but for larger files, per kilobyte reporting is probably a nuisance. In our experiments with the modified version of the program, we used the program to download a large file.[5] We then compared the average amount of time (averaged over 100 executions) taken by the modified version of the program against the original version. The comparison revealed a performance improvement of 2.6× as compared to the original version of the program. Six new bottlenecks are reported for axel when multiple execution profiles are analysed. All six of the bottlenecks correspond to the main loop inside the *axel_do(. . .)* method. The reported basic blocks correspond to conditionals that belong to non-deterministic events that deal with making a connection to the server.

Table 4.1: Number of performance bottlenecks for a single execution of the program

Application	Profiles	Bottlenecks	Threshold
pbzip2 - original	1	1	5%
pbzip2 - workload	1	2	5%
axel - original	1	4	5%
Apache httpd start	1	3	5%
Nmap - original	1	6	5%

4.5.3 Nmap

Our experiments with Nmap report six bottlenecks. The bottlenecks mostly belong to basic blocks inside of library routines for the scripting engine used by Nmap. The code regions belong to library routines for the programming language *Lua*, which is used for Nmap’s scripting engine. These code regions include a loop performing the hash inside the hashing library function and conditionals belonging to the String library routines of the Lua programming language/environment.

[4]The file to be downloaded is served by a local machine to avoid perturbation of the results by wide-area network delays.

[5]The downloaded file is a CD image (iso) file containing the Debian operating system, and it has a size of 627 MB.

Table 4.2: Number of performance bottlenecks reported for multiple executions of the program

Application	Profiles	Bottlenecks	Threshold
pbzip2 - original	100	4	5%
pbzip2 - workload	100	5	5%
axel - original	100	10	5%
Apache httpd start	100	3	5%
Nmap - original	100	11	5%

When 100 execution profiles are recorded and analysed, the number of reported potential bottlenecks almost doubles. Inspection of the five newly reported possible bottlenecks include a few more basic blocks in the Lua programming language's library routines and two basic blocks belonging to the *traceroute* system utility. Overall, while using a threshold of 5% we did not identify any bottlenecks in Nmap.

4.5.4 Apache httpd web server

Next, we applied our tool to the latest version of Apache httpd web server (version 2.4.16). Our experiments were performed on the startup routine of the web server. These experiments report three bottlenecks at a pruning threshold of 5%. The three reported bottlenecks belong to basic blocks inside a string utility used by Apache. The routine is invoked multiple times when reading configuration settings from the configuration files on disk. The routine itself probably has little opportunity for further optimisation, since the routines perform basic string operations. However, the experiments illustrate the effectiveness of our tool in successfully identifying these time-consuming portions of the program.

The experiments performed using our tool serve to validate the effectiveness and usefulness of our approach. The key insight gained from these experiments is that our approach succeeds in identifying locations in the program where it spends a significant portion of its execution time. The locations are identified at a fine-grained level, making it possible to inspect and improve program performance without much effort. For deterministic performance bugs, a single execution profile should usually suffice. However, to unearth non-deterministic performance bugs multiple executions of the program, and different inputs may be needed. Although our approach makes an effort to exercise different interleavings of the program threads, there are no guarantees that this will work in practice. Further investigation in the direction of scheduling is outside the scope of our work and was not explored.

4.5.5 Performance

The performance overhead of both the on-line and off-line analysis of our approach is presented in Table 4.3. These experiments were performed on a machine with an

Intel core i7 3.5GHz processor, having 16 GB of memory, and running the desktop version 12.04.5 of Ubuntu operating system. The selected applications range in size from a few thousand lines of code to over 400,000 lines of code in the case for Nmap. The first phase of our approach has a higher performance overhead than the second phase. The second phase analyses the profiling data collected in the first phase and is very fast, as can be seen from the fourth column in Table 4.3. The advantage of our two-phase approach is that once all the data is collected in the first phase, the user is free to conduct more off-line analysis. The collected data are never invalidated by the off-line analysis. Since the off-line analysis is fast, the user has the advantage of experimenting with different threshold values during the second phase. To fairly judge the performance of the off-line analysis engine, 100 profiling runs were analysed, to ensure that the data are reproducible.

Table 4.3: Performance overhead of the on-line profiling and off-line analysis routine.

Application	Profiles	Offline Analysis	Online Profiling	Threshold
pbzip2	100	0.100 seconds	28.71x	5%
Apache Httpd	100	0.591 seconds	3.38x	5%
axel	100	0.0464 seconds	1.97x	5%
Nmap	100	1.07 seconds	7.01x	5%

The overhead incurred by the on-line analysis is presented in the last column of Table 4.3. The overhead is averaged over 100 executions to allow a fair analysis of the implementation. The overhead is slightly larger in the case of pbzip2, but it is orders of magnitude less for larger applications like Apache Httpd and Nmap. There are a couple of reasons for the high overhead for pbzip2. First, since pbzip2 is a smaller application, the set up time required by our framework to load and start the application adds significant overhead to the execution time. Second, pbzip2 uses many conditionals for its size, and the number of times that these conditionals are executed is fairly high, in part because of the sorting and compression algorithms that are used by pbzip2. Since our profiling engine collects data at each conditional branch, this setup leads to higher overhead for smaller programs with more conditionals. Experiments using the larger applications Apache and Nmap serve to validate our claim that the overhead remains reasonable as applications grow in size.

4.5.6 Threats to validity

There are a number of issues to consider when dealing with performance bugs for concurrent programs. First, concurrent programs are inherently non-deterministic, and the number of possible thread schedules is astronomically large for most medium to large-scale applications. If the program is not properly synchronized and access is not properly guarded, then different thread schedules may lead to different program behaviour — regardless of the supplied input. Second, performance bugs in concurrent programs behave the same way as concurrency bugs, i.e., they appear only under rare circumstances. These rare circumstances can be a rare interleaving of the program's threads or a particular program input, that may lead to contention for resources. The approach described here attempts to discover performance bugs from execution profiles that are collected at runtime. If a performance bug does not appear during the recorded program executions, then it will be missed by our approach. The tool presented in this chapter tries to influence the scheduler by delaying random memory accesses, but there are no guarantees that this will cause a rare resource contention scenario to surface.

The technique presented here is not prone to false positives because it reports events that are above the user-specified threshold. However, these events could very well be legitimate program statements and it may not be possible to remove them for certain programs. The use of a threshold value means that our approach relies on the user's intuition of what they perceive to be a performance bug. For example, certain users might not consider a basic block that takes 5% of the total program execution time to be a performance bug. The approach only reports information according to the criteria that are specified by the programmer, and is based on profiling information collected during program executions.

4.6 Related work

Apart from some work in the high performance computing community, performance bugs have largely been ignored in previous research on software defects. Many research efforts on software defects focus on functional rather than performance bugs. A recent empirical study by Jin et al. on performance bugs presents a good overview of a wide range of bugs collected from real-world applications. The study provides guidance for performance bugs detection; the authors explore a rule-based performance bug detection technique to uncover previously unknown performance problems [56].

Most work on detecting performance bugs focuses on identifying code locations that take a long time to execute in certain program executions. Profilers like oprof and gprof [46, 71] periodically sample the program counter during execution of the program. These tools then propagate the samples through the call graph during post-processing to arrive at estimates on how much of the total running time was spent in each function of the program. Using such profilers is the standard way to find optimisation opportunities for improving the performance of a program. Recently, profiling techniques involving dynamic analysis to detect excessive memory usage have been proposed. These include techniques for optimising the creation of similar data structures over the lifetime of a program [136], as well as techniques that track the life-time of objects to uncover run-time inefficiencies in programs [138]. Other profiling techniques work on recording memory access behaviour of programs that use recursive data structures [111] by capturing the run-time behaviour of the individual instances of structures such as lists and trees. More recently, work by Nistor et al. [95] presents a technique for detecting code loops whose computation has repetitive and partially similar memory-access patterns across loop iterations. Nistor et al. suggest that performing such repetitive work is likely to be unnecessary and may be performed faster by caching the repetitive behaviour. Mitchel et al. [83] focus on analysing data structures at runtime to find their execution cost. Xio et al. track data structures to perform post-mortem analysis of the collected information and suggest improvements to the design of data structures [135] . Pradel et al. [108] suggest a performance regression testing technique for thread-safe classes. The technique works by generating multi-threaded performance tests and comparing two versions of a class to detect performance gains.

A body of research work also focuses on techniques for identifying the root cause of a performance bugs within applications. An example is a tool like *X-ray* [5]. X-ray is built using techniques of dynamic information flow analysis and deterministic record and replay. X-ray uses performance summarisation techniques to automatically diagnose the root causes of performance bugs. The tool assigns costs to each basic block within the program using metrics like CPU utilisation or network activity (by using binary instrumentation). The tool then records intervals of program execution to analyse. The execution recorded by X-ray may then be replayed during analysis.

More recently there has been research work carried out to suggest fixes for performance bugs based on dynamic analysis. These include tools, like MemiozeIt [127], which searches for memiosation opportunities within applications. The tool performs dynamic analysis for identifying methods that repeatedly perform the same computation. These computations are identified by comparing the input and outputs of method calls. MemiozeIt then suggests caching opportunities in profiled code to improve program performance along with hints on how to implement the memiosation. After ensuring their effectiveness, the fixes suggested by MemiozeIt may be applied by the programmer. Similarly the performance tuning framework called AutoTune [80] offers to automate both analysis and optimisation for performance and energy efficiency tuning in parallel applications. Autotune is based on Periscope [11] — a tool for online distributed performance analysis that detects performance bottlenecks in parallel applications. The tuning framework performs instrumentation and static analysis of the program source code. The tool's search for a tuned versions involves iteratively selecting plugins according to the tuning objectives (such as program runtime or energy consumed). As the tuning objective for a single node heterogeneous manycore architecture; the tuning of a high-level pipeline pattern as explained by Bajrovic et al. [7] is presented by the authors. The process involves continuously selecting plugins based on their tuning strategy and transforming the tuning parameters, followed by an experimental evaluation. Autotune generates a tuning report to present the recommended tuning actions in the form of changes to the source code that must be incorporated into the application by developers.

Profiling techniques for detecting performance bugs that utilise multiple program runs include techniques like algorithmic profiling [145]. Algorithmic profiling works by automatically determining an approximate cost function based on multiple program runs. Goldsmith et al. focus on calculating empirical computational complexity of programs by running a program with several inputs, and fitting the costs to a curve for determining performance as a function of workload size [45]. Maddukrishna et.al [85] present an automated approach for extracting task-based performance information from OpenMP programs to diagnose performance problems.

Our work combines the approach used by state-of-the-art profilers and those approaches that analyse multiple executions of the program for detecting performance bugs. Our approach can identify those basic blocks where program executions spend most, but not all, of their time i.e., the non-deterministic cases. Furthermore our approach is not limited to any specific software or hardware architecture and offers a wider scope for detecting performance bugs in applications.

4.7 Extensions

Our experimental results reveal certain aspects that would benefit from further investigation. First, the main factor contributing to our high overhead is the number of conditional branches that are executed in a single program run. The approach relies on collecting profiling information for basic blocks at the boundary of conditional branches. Thus, the overhead incurred by our approach is directly proportional to the number of conditional branches executed for some program input. One way to reduce the overhead would be to collect profiling information at a higher granularity than a basic block. The granularity could be that of defined paths through the program, e.g., using ideas from the path profiling algorithm presented by Ball et a. [9]. Instrumentation need not occur at every conditional branch. Implementing programs at a higher granularity, or leveraging a statistically fair sampling strategy might help reduce the overhead of the first phase in our approach. Furthermore, if profiling information at the class or method level is available, then it could be used to mask monitoring the execution of those parts of a program that contribute less to the program's execution time.

Second, our approach currently collects profiling information about the hotness, the minimum, maximum, and the average amount of CPU time spent in a basic block. The approach would benefit from collecting more information at these code regions, including information regarding cache misses and stall cycles.

4.8 Chapter summary

Performance bugs are hard to fix, and the tools to aid programmers in detecting these bugs are scarce or lacking in functionality. This chapter presents an approach for detecting and localising performance bugs. We present a prototype implementation of the approach to conduct an experimental evaluation of the system. The tool profiles and detects performance problems in applications using profiled execution information. Our evaluation of the approach shows that the tool indeed detects performance problems in real-world applications, and that we are able to localise the problem to the level of a basic block.

We use binary rewriting to collect data about a program's execution. Although binary instrumentation incurs a high overhead, it nevertheless allows the use of this technique in realistic settings. The overhead is probably too high to allow the system to be used by default, but the overhead is low enough to allow its use as soon as a problem is suspected.

Our decision to use basic blocks bounded by conditional branches as analysis points in the prototype is based on the fact that current support for monitoring execution behaviour is limited, especially for multi-threaded programs. Other options (e.g., replacing binary instrumentation by a compiler framework to track conditional branches that are executed, and directly using the processor's program monitoring unit) are worthy of further investigation. However, the difficulties to obtain valuable data about a program's execution point to the need for better program monitoring units. Given the proliferation of multi-core processors and the ability to provide additional functionality in processor implementations, architects of future processors should pay more attention to the needs of tools to help software engineers isolate and understand program performance defects.

5
Identifying root causes for performance bugs

This chapter presents the detailed description of a methodology for identifying the root causes of performance anomalies in concurrent software. As part of the framework, we suggest improvements and extensions to the work presented in Chapters 3 and 4. The aim of the framework is to combine and improve upon ideas from the work presented in those chapters. The framework can identify the probable root cause of performance anomalies in parallel applications. The framework is can detect performance bugs using techniques borrowed from Chapter 4 with some improvements. Once a performance bug is detected, the infrastructure can find the probable root-cause of the detected bug by using a modified version of the approach presented in Chapter 3.

5.1 Overview

Researchers are mostly in agreement with regards to the major steps involved in troubleshooting/debugging of anomalous program behaviour. The first step towards debugging a bug that is triggered is to detect that the program has exhibited anomalous behaviour. Once a bug is detected, the second step involves identifying the root-cause for the behaviour, and finally as the last step, the bug may be patched manually or an automatic patch may be generated that could be applied to fix the problem.

Most prior research work as well as the work presented in Chapter 4 explores ideas for dealing with the aforementioned first step in the debugging process, i.e., detecting that the program has exhibited anomalous behaviour. However, working towards the first step of detecting anomalous behaviour solves only one half of the debugging problem. The second half identifying the root cause for such an anomaly is yet to be fully explored. The state-of-the-art in performance debugging is that once some anomalous program behaviour is detected, programmers must manually inspect the part of program that exhibits the anomalous behaviour. The work presented in Chapter 4 can aid in detecting anomalous program behaviour, and provides programmers with a small part of the code to inspect. The code to inspect is in the form of one or more basic blocks. The reported basic block(s) are symptom blocks, i.e., they are where the anomalous behaviour of the program manifests itself and makes an impact on the execution behaviour of the program. Programmers must then try to infer the root cause for the behaviour by inspecting and debugging starting at the identified symptom basic blocks. However, the root cause for a performance anomaly may be at a code location that is far away from its symptom (i.e., the place where the anomaly makes a visible impact on the performance of the program), and may in fact be in an entirely different but related component of the software system. As an example, consider the situation where we identify a symptom for poor program performance in a basic block where the program spends most of its execution time. However, the identified basic block may not contain the root cause for the observed behaviour, i.e., the reason why the program exhibits the anomalous behaviour.

The framework presented in this chapter combines the ideas discussed in Chapters 3 and 4 for building a system that is able to identify the root causes of performance anomalies in concurrent software. The work on performance anomaly detection can identify those basic blocks where some executions spend most, but not all, of their time. Detecting concurrency bugs employs dynamic binary instrumentation to identify changes in the control flow path. Here we present a framework that improves upon the two approaches presented in the aforementioned chapters to identify the root causes of performance anomalies. The key insight is that if we can identify code regions where the program spends most, but not all, time on some runs, then, we may identify the root cause of this anomalous behaviour by looking at the control flow path on the two different executions. The idea is based on the intuition that the control flow path followed by the program in executions where the anomalous behaviour manifests itself may not be the same as the control flow path followed by other executions. Thus, we may be able to identify the probable root cause by comparing the control flow paths. We show in Chapter 4 that these performance anomalies may only appear on rare occasions, i.e., only for some program inputs. Therefore, comparing the control flow path from an anomalous execution to any other execution where the anomaly does not appear may reveal the part in the program that is responsible for the behaviour.

5.2 Background

Debugging performance problems in large-scale software is a challenging task and has recently become the focus of much work [26, 56, 61, 80, 94, 143, 144]. There are a number of research studies on performance bugs that provides an overview of performance problems in programs, and these studies aid in developing a deeper understanding of the different types of performance bugs that appear in released software. These studies provide an excellent synopsis of the types of performance bugs that exist in released software, the sources that introduce these bugs, and some root cause analysis. Furthermore, these studies help in identifying patterns that may assist in performance bug analysis. The aforementioned research studies reveal that performance bugs are indeed a real problem, and they are prevalent in most modern software.

Most existing research approaches, including ours (from Chapter 4), focus on identifying the bottleneck symptoms in software systems. Although there are a large portion of the approaches in the literature that are helpful in aiding the debugging process for performance anomalies, however they are unable to assign blame to specific code segments for causing the behaviour. Profiling tools like VTune [112] use hardware performance counters to determine program regions that incur a large number of cache misses or branch mispredictions. The profiled information relates to frequently executed loops as well as those parts of the program that account for the most misses. The information, although useful, cannot uncover the reason for the cache misses, or why a large portion of time spent in a code region. For example, the tool may report that the program spends a large portion of its time inside some loop of a compression algorithm. However, timing information alone is less likely to be useful for any optimisation since the compression algorithm may have already been heavily optimised. As an example, for the experimental results with pbzip2 presented in Chapter 3, most of the basic blocks that get reported (when log pruning is turned off) belong to sorting and compression algorithms. Similarly, most libraries dealing with string manipulation are highly optimised but still take a large portion of the execution time for some applications. We argue that execution time, although an important aspect, is not sufficient alone to determine whether a code region is a potential performance bottleneck. This is because most commercial applications are highly optimised, and merely identifying hot code paths or regions that take a significant fraction of the overall execution time may not be enough.

The execution time information combined with cache misses and stall cycles may present a better picture of the overall behaviour of the program. The reason cache misses are an important piece of information is because processors can consume data much faster than off-chip memory can supply it. Although on-chip caches and prefetching can help somewhat in this regard, programs still spend a significant fraction of their time waiting for memory. The problem is exacerbated by multicore chips because they increase the demand for data and may cause even more misses due to contention for shared data. There are a number of reasons for cache misses [49, 105]: *cold misses* are cache misses when a location is read for the first time; *capacity misses* are caused when the cache cannot contain all the blocks needed during program execution, and blocks have to be replaced and retrieved again when needed; *conflict misses* occurs in a set-associative or direct-mapped cache when multiple blocks compete for the same set (or cache line). Similarly stall cycles are an important aspect in determining program performance because it too affects CPU execution time. We deems cache misses to be a critical component beside stall cycles and CPU execution time for improving performance of a system. Together, the three aspects including cache misses, stall cycles, and execution time may provide a more complete picture of the performance of a software system.

Recently a research study by Yin et al. reported that up to 20% of the

```
class ExpandoMetaClass {
   private boolean initialized;
   synchronized void initialize() {
      if (!this.initialized) {
         this.initialized = true;
      }
   }
   boolean isInitialized() {
      return this.initialized;
   }
}
```

(a) Initial version of the class that is not thread-safe, since access to *initialized* is not protected.

```
class ExpandoMetaClass {
   private boolean initialized;
   synchronized void initialize() {
      if (!isInitialized()) {
          setInitialized(true);
      }
   }
  synchronized boolean isInitialized() {
      return this.initialized;
   }
   synchronized boolean setInitialized
      (boolean b) {
         this.initialized = b;
   }
}
```

(b) Patch is applied to fix the functional bug, introduces a performance bug in October 2007.

```
class ExpandoMetaClass {
   private volatile boolean initialized;
   synchronized void initialize() {
      if (!isInitialized()) {
          setInitialized(true);
      }
   }
   boolean isInitialized() {
      return this.initialized;
   }
   boolean setInitialized (boolean b) {
         this.initialized = b;
   }
}
```

(c) Two years later, the performance bug is fixed in September 2009 by replacing the synchronized methods with a volatile variable.

Figure 5.1: How a performance bug was introduced in Apache Groovy [50] while fixing a functional bug.

misconfigurations submitted for developer support are the ones that result in severe performance degradation [141]. Further, Attariyan et al. claim that this reported number of misconfigurations is significantly higher than that reported by Yin et al [5] . The main reason cited for such a high number of reports is that troubleshooting performance anomalies focuses on *what* (i.e., the symptoms) behaviour is exhibited by the program and not *why* (i.e., the root-cause) the program exhibits such behaviour. An example of a real-world performance bug is presented in Figure 5.1 [1]. This example is interesting because it shows how a

[1]We gather the details for this bug from Pradel et al. [108].

performance bug is introduced while trying to fix a functional bug in the program. Initially the code is not thread-safe, since access to the *initialized* flag field is not protected. Developers make an attempt to fix the thread-safety issue by adding two synchronized methods to guard access to the shared flag. However, this change has a significant negative impact on performance. Two years later, the performance problem was localised to the same code. This time the developers come up with the right fix by making the field volatile to avoid unnecessary synchronisation. An interesting comment to note here is that the developers did not treat the performance issue as a real bug. One of the developer commented: *"I reduced the priority to 'major', because this is not really a critical bug. Also it is no real bug because it just works. So I changed it into 'improvement'"*.[2] The comment proposes that performance bugs are not real bugs but rather an improvement to software. Perhaps this attitude towards performance bugs may not be justified but goes to show that these bugs do not receive the same recognition as functional bugs. Consequently, this lack of attention translates to a lack of work to address performance bugs in the same way as their functional counterparts. The example in Figure 5.1 reveals just one source of performance bugs (i.e., over synchronisation) and goes on to show that performance bugs may remain in software for a long time (almost two years in this case).

The task of identifying the root cause of performance bugs becomes a challenging activity when dealing with large-scale software systems. Root cause identification becomes especially challenging in a concurrent setting, where program behaviour is non-deterministic. This non-determinism may not only be based on program input but also on the order in which program threads are scheduled. The number of possible thread schedules are large in practice for any reasonably sized software system. A rare interleaving of program threads could cause a schedule that is hard to reproduce. In practice, exercising every possible interleaving is a computationally intractable problem, and is perhaps one of the reason why rare software bugs remain dormant for a long time. Therefore, once a rare thread interleaving is observed, a testing infrastructure should be able to satisfy queries regarding the program behaviour for the observed execution. Our infrastructures share this objective, i.e., to be able to satisfy queries regarding observed program behaviour. Next, we present the detailed design of our infrastructure.

5.3 Design and implementation details

Our infrastructure focuses on the first two steps of the debugging procedure outline in Section 5.1, i.e., to detect that the program has exhibited anomalous behaviour and then to find the root-cause for that behaviour. The approach can localise the anomalous behaviour to a basic block. The high-level design is depicted in Figure 5.2. The process of identifying the root-cause follows a two-phase approach.

[2]Comment by a developer on bug 3557 from the Groovy database.

The first phase is concerned with dynamic analysis to collect profiling information about execution behaviour. The second phase involves a post-mortem, offline analysis of the collected information to identify deviant behaviour and then discern the root cause of the problem. The offline phase first identifies the bottlenecks within the program, followed by a step to find the probable root-cause for each of the identified bottleneck region. The confirmed bottleneck regions along with the probable root causes identified by the infrastructure are then reported to the user for inspection. A detailed description of the online analysis steps are presented next, followed by a description of the offline analysis phase.

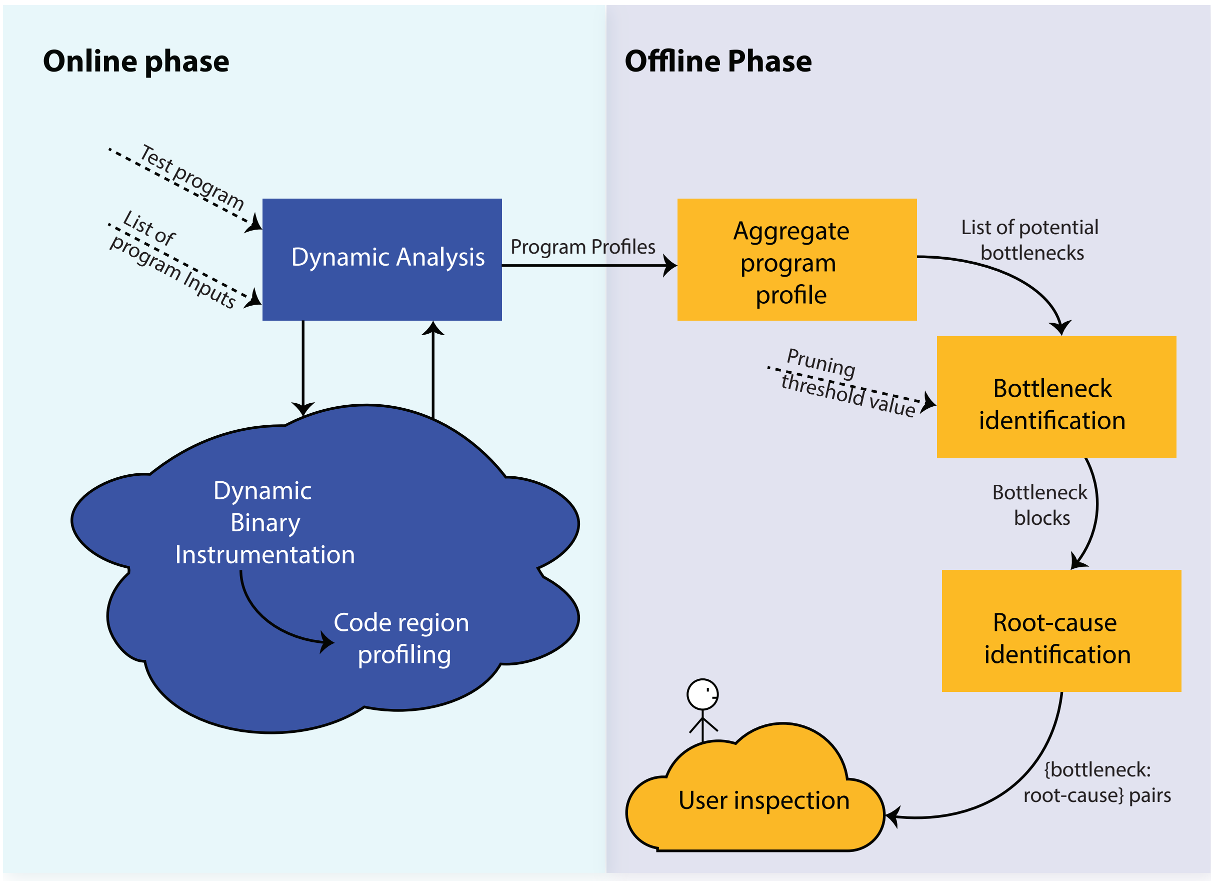


Figure 5.2: The online profiling phase is followed by an offline analysis of the information to determine root causes of anomalies.

5.3.1 Online analysis

The online analysis technique builds on our work from Chapter 4. Parts of the online technique for root-cause detection are similar to the approach presented in Section 4.3, Algorithm 3. As in previous technique, the infrastructure must collect online profiling information to identify the bottleneck code regions first. The procedure for the online profiling technique employed here is laid out in Algorithm 5. The online phase involves collecting timing information for every code region. Although, non-functional properties like timing information are important and provide a good overview of program performance, however it is not the only piece of information that may reveal non-functional problems in a program. To this end the proposed infrastructure also collects profiling data regarding cache misses (Line 11 of the algorithm) and stall cycles (Line 10 of the algorithm) as well, apart from timing and hotness information for code regions that get executed. The algorithm calculates a rolling average for both the stall cycles and cache misses per code region to average out any outliers. At the end of the online phase, the recorded profiling information may be stored permanently to be utilised during the post-mortem analysis phase.

```
input  : $PUT$ - Program under test
output: $profile$ - A log file containing hotness and timing information per
        basic block
1  $instrument$(PUT) ;
2  while PUT.isAlive do
3      for each CR in PUT do
4          CR.execFreq+=1;
5          execTime=CR.exitTime-CR.entryTime;
6          if CR.execFreq >1 then
7              CR.avgTime= (execTime +
                (CR.avgTime*CR.execFreq))/(CR.execFreq+1);
8              CR.minTime = min(CR.minTime, execTime);
9              CR.maxTime = max(CR.maxTime, execTime);
10             CR.stalls= (stalls + (CR.stalls*CR.execFreq))/(CR.execFreq+1);
11             CR.misses= (misses + (CR.misses*CR.execFreq))/(CR.execFreq+1);
12         end
13         else
14             CR.avgTime = CR.minTime= CR.maxTime = execTime;
15         end
16     end
17 end
18 $WriteProfileLog$(PUT);
```

Algorithm 5: The online profiling phase of the programs under test.

The work presented in Chapter 4, incurs a significant amount of performance overhead during its online analysis phase. The proposed infrastructure could improve upon the performance overhead incurred during the online analysis phase by improving upon the approach and employing better tools. There are two main source of performance overhead associated with the approach presented in Chapter 4. First, the binary instrumentation framework used during the online analysis phase, and second, the our data collection procedure for each basic block. The proposed infrastructure may avoid the overhead that is associated with binary instrumentation by exploiting the information readily available via hardware performance counters for profiling the required information. The implementation to accommodate these changes should be quite straightforward. Profiling data may need to be collected at clearly marked code regions. As in our previous approaches, profiling at the level of a basic block may be too fine-grained for some applications. As an example, applications involving compression or sorting algorithms usually have a large number of conditional jumps (which end a basic block in our case), and these applications may suffer a high overhead because of such fine-grained profiling. The online profiling overhead may be reduced by profiling at a higher granularity by borrowing ideas from Ball et. al [9]. Further, we propose the use of a lightweight instrumentation framework such as the Bursty tracing framework [52], or some low overhead process tracing utility like the Linux Trace Toolkit [137] that already exists to collect the profiling information. Profiling information may be collected directly from the processor's process monitoring unit by using hardware performance counters. The runtime overhead of using hardware performance counters is almost negligible since the information is readily available by accessing a register of the processor.

5.3.2 Offline phase

The offline analysis phase uses the profiling information collected during the online analysis phase. We report two pieces of information in this phase: First, we report the identified bottleneck code regions and, second, the probable root-cause for the identified regions. The approach works in two steps to report on the two pieces of information. Next, we explain the details of our bottleneck identification procedure.

Bottleneck identification step

The algorithmic details of the first offline analysis task for identifying the bottleneck code regions are laid out in Algorithm 6. The algorithm follows most of the steps outlined in Section 4.3, Algorithm 3 with some modifications to include profiling information regarding cache misses and stall cycles, along with the average time and statistical range of the minimum/maximum time spent in the code region. The algorithm works by generating an aggregate profile from the recorded executions of the program. The aggregate profile is a representative profile of the observed behaviour of the program. The aggregate program profile contains the maximum amount of average time spent in a code region (Line 16 of Algorithm 6), the maximum hotness value for the code region, i.e., how many times was the code region executed in a single execution (Line 17), the maximum of the average cache misses (Line 20), and stall cycles (Line 21) that were ever recorded for the code region in all the executions, and the absolute minimum time (Line 18) that was ever spent in a code region along with the absolute maximum time (Line 19).

As in the approach of Chapter 4 the algorithm makes use of threshold value for eliminating false positives. The algorithm requires three different threshold values: first, a threshold value for the amount of time consumed by a code region as a percentage of the total execution time, second for the number of percentage cache misses and, third for the number of stall cycles. The threshold values must be

```
input  : $programProfiles$ - List of program profiles collected in the online
         phase.
input  : $timeThreshold$ - Threshold percentage value of total execution time.
input  : $missThreshold$ - Threshold percentage value of total cache misses.
input  : $stallThreshold$ - Threshold percentage value of total stall cycles.
output: $bottleneckCRList$ - List of potential performance bottlenecks.
totalExecTime = $readProfile$(profileLog) ;
totalMisses = $readProfile$(profileLog);
totalStalls = $readProfile$(profileLog);
eThreshold ← (totalExecTime*eThreshold)/100;
crList ← $empty$ ;
while more profileLog exist do
    $read(profileLog)$;
    for each CR in $profileLog$ do
        if crList.exists(CR) then
            prevAvg = crList.CR.avgTime;
            prevFreq = crList.CR.execFreq ;
            prevMin = crList.CR.minTime;
            prevMax = crList.CR.maxTime;
            prevMisses = crList.CR.misses;
            prevStalls = crList.CR.stalls;
            crList.CR.avgTime = max(CR.AvgTime, prevAvg);
            crList.CR.execFreq = max(CR.execFreq, prevFreq);
            crList.CR.minTime = min(CR.minTime, prevMin);
            crList.CR.maxTime = max(CR.maxTime, prevMax);
            crList.CR.misses = max(CR.misses, pervMisses);
            crList.CR.stalls = max(CR.stalls, pervStalls);
        end
        else
            crList.add(CR);
        end
    end
end
for each CR in crList do
    CRTSig ← CR.execFreq * CR.avgTime ;
    CRRange ← CR.maxTime - CR.MinTime ;
    CRStallsSig ← CR.execFreq * CR.stalls;
    CRMissSig ← CR.execFreq * CR.misses;
    if CRSTig >= eThreshold and CRRange >= eThreshold then
        if CRStallsSig >=stallThreshold or CRMissSig >=missThreshold then
            $bottleneckCRList.add$(CR);
        end
    end
end
```

Algorithm 6: Finding the bottleneck code regions within a program.

specified as a percentage of the total in each case. For example, in the case of the threshold value for time, it is the percentage of the total execution time consumed by the program. The algorithm uses the three threshold for reporting bottlenecks (Lines(29 to 35) for the next step of root-cause identification. The algorithm uses timing information in two ways: First, the product of the number of times that a code region is executed and the average maximum time it took for this bottleneck to execute must be higher than the specified threshold for time, and second, the statistical range of the minimum and maximum time spent in a basic block must be greater than the threshold for time. The check for statistical range of the time spent, helps in identifying those code regions where the time spent is higher (or lower) for some inputs/schedules of the program as compared to others. After these two timing conditions are met, the algorithm checks whether the consumed time was because of cache misses or stall cycles before marking the code region as a bottleneck. We regard the time spent in a code as the main condition for reporting bottlenecks, because both cache misses and stall cycles are bound to affect timing of the program. If the cache misses and stall cycles are both low, then the code region is not marked as bottleneck. The intuition is that if the cache misses and stall cycles are low, then the high amount of time spent in the basic block may be an inherent property of the algorithm and cannot be helped. The identified bottleneck code regions are fed into the next step of finding the root-cause for each of these bottleneck regions.

Root-cause identification

Once the list of potential bottlenecks is identified, we generate the end user report. The user report contain two key pieces of information: First, the identified bottleneck regions, and second, for each code region we report the probable root cause of the bottleneck. The probable root cause points to a code location where the control flow path deviates from an execution where the bottleneck does not appear. The approach ensures that both of these program profiles are available, otherwise no bottleneck is reported.

The details of our root-cause identification procedure are presented in Algorithm 7. Concretely, to find the bottleneck profile, the approach utilises a program profile that has the largest value for the recorded maximum time for the identified bottleneck region from Algorithm 6. The action of selecting the bottleneck profile is performed at Line 2 of Algorithm 7 by invoking the *getBottleneckProfile* procedure.

Once the bottleneck program profile has been selected, it is then compared against all recorded profiles to find one that may contain the probable root cause for the selected bottleneck. The profile containing the probable root-cause is referred to as *deviantProfile* in the algorithm. The deviant profile is identified by *compareCF* procedure of the algorithm. The procedure identifies the deviant program profile by comparing the control flow path of bottleneck profile against the path for other program profiles, up to the reported bottleneck code region. The profile that has recorded a control flow that deviates from the bottleneck profile is selected as the deviant profile and is believed to contain the probable root-cause for the bottleneck. The root-cause for the bottleneck is the point where the two control flow paths diverge. The intuition is that rare bottleneck blocks may also affect the control flow path of the program. Algorithm 6 ensures that only those bottlenecks are reported that do not appear on all of the program executions, i.e., they appear for only some program inputs or thread interleavings, but not all.

The bottlenecks verified in this way are then reported to the programmer for inspection, along with the probable root-cause. The root-cause is reported to be the conditional branch where the control flow path of the program deviates for the two execution profiles being compared. The idea is that a program may follow different control flow paths for different inputs of the program, or it may do so because of a harmless data race within the program under test. In both the aforementioned cases, if the changes in control flow path have an effect on program performance, and this effect is significant enough to be detected in the first step of the offline analysis, then it must be reported to developers for inspection. The final report contains the two profiles with the deviant control flow paths along with the bottleneck code region and a probable root-cause for the bottleneck in the form of a conditional branch where the two control flow paths deviate. The reported bottleneck and root-cause information may then be mapped to source code to confirm the truth of the reported bottleneck.

5.4 Related work

The issue of root-cause detection for performance bugs has received considerable attention recently. Hardware performance counters have been utilised for detecting functional bugs in programs including detecting data races [47, 123] as well as atomicity and order violations [4]. However, there has been little use of hardware performance counters for performance bug detection. Research work that utilises performance counters for detecting performance problems include variational path profiling [104] for discovering performance optimisation opportunities, and DProf [105] - a tool for locating cache performance bottlenecks using profiling data.

The work that is most closely related approach to the technique presented in this chapter is variational path profiling [104] for finding performance optimisation opportunities. Variational path profiling attempts to find the acyclic control flow paths that vary the most in execution time (i.e., the time it takes to execute each occurrence of that path). The technique samples the time it takes to execute frequent paths using hardware performance counters. The idea is that acyclic program paths that have the highest net variation in their execution time are candidates for significant optimisation opportunities. The net variation is determined by finding the smallest execution time for a path and then summing up the additional time it took to execute each invocation of that path over its smallest execution time. A path with high net time variation implies that there

```
input  : $profileLogs$ - List of program profiles from Algorithm 6.
input  : $bottleneckCRList$ - List of bottleneck code regions Algorithm 6.
input  : $normalRange$ - Normal time range value from Algorithm 6.
output: $bottleneck : rootCause$ - A tuple of bottleneck code regions and the
        probable root-cause.
1  for each bottleneckCR in bottleneckCRList do
2      bottleneckProfile = getBotlleneckProfile(bottleneckCR) ;
3      report.bottleneckProfile ← bottleneckProfile;
4      while more profileLog exist do
5          currentProfile ← readNextProfile();
6          deviant, deviantPoint = compareCF(currentProfile, bottleneckProfile);
7          if deviant then
8              report.bottleneck ← bottleneckCR;
9              report.rootCause ← deviantPoint;
10             report.deviantProfile ← currentProfile;
11         end
12     end
13 end
   procedure getBotlleneckProfile(bottleneckCR)
       maxTime ← bottleneckCR.maxTime ;
       for each profile in ProfileLogs do
           currMaxTime ← profile.bottleneckCR.maxTime;
           if maxTime <= currMaxTime then
               maxTime ← currMaxTime;
               bottleneckProfile ← profile;
           end
       end
       return $bottleneckProfile$
   end procedure
   procedure compareCF(currentProfile, bottleneckProfile)
       deviant ← false;
       for each bBranch in bottleneckProfile, currBranch in currentProfile do
           if bBranch != currBranch then
               deviant ← true;
               deviantPoint ← currBranch;
           end
       end
       return $deviant, deviantPoint$
   end procedure
 $Report$(report);
```

Algorithm 7: Identifying root causes for the bottleneck code regions identified in Algorithm 6.

are certain executions of that path that took significantly more time to execute than other executions of the program. They argue that every dynamic invocation of a path should be able to only take the minimal sampled execution time assuming that stalls on the time consuming invocations can be removed with optimisations.

DProf [105] aims to aid programmers in eliminating cache misses in a program. DProf uses performance monitoring hardware to accumulate traces of the program's references to memory addresses. It then categorises the cache misses as either associativity misses, true sharing misses, false sharing misses and capacity misses [3]. After the categorisation process, DProf attempts to identify the actual types of data participating in the cache misses and presents the data to programmers in a number of pre-defined views for different representations.

Other interesting research work on detecting performance problems include magpie [10] - an online modelling service for performance debugging and anomaly detection in distributed systems, X-ray [5] - a performance summarisation technique for automatically diagnosing the root-causes of performance problems, and the work of Aguilera et. al [1] on performance debugging for distributed systems of black boxes. Magpie proposes a modelling service which collates detailed traces from multiple machines, extracts request-specific audit trails, and constructs probabilistic models of request behaviour. Magpie employs black-box instrumentation of the program binary and end-to-end tracing to track aggregate statistics as well as an individual request's path through the system. X-ray uses a performance summarisation technique to automatically diagnose the root-causes of performance problems. It focuses on attributing performance issues to specific root causes in the form of configuration settings and program inputs. X-ray uses metrics such as CPU utilisation, file system activity, or network usage to measure performance costs. It then produces a list of root causes ordered by the likelihood that each cause has contributed to poor performance during the monitored execution. X-ray's performance summarisation technique attributes performance costs to each user-level instruction and all system calls that are executed by the application. It uses dynamic information flow analysis to associate each such event with a ranked list of probable root causes and then summarises the cost of each root cause over all events. X-ray produces a list of root causes ordered by performance costs. The work of Aguilera et. al seeks to infer causal paths between application components of a distributed system and attributes delays to specific nodes. The authors tackle the problem by passively tracing communication between different nodes of the distributed system and some offline analysis of the collected traces.

Our approach differs from X-ray in a number of ways. Our approach attempts to identify control flow paths through the program that are responsible for the anomalous program behaviour, whereas, X-ray focuses on attributing performance issues to configuration settings or program inputs. Our approach aims to combine ideas from both variational path profiling and Dprof. Similar to our approach,

[3]Details of the different categories of misses are laid out in [49, 105].

variational path profiling bases its decision on execution time. The technique used in variational path profiling records the execution time for frequent acyclic control flow paths using hardware performance counters. This allows variational path profiling to identify locations in the program code where there are significant optimisation opportunities for speedup by focusing on the control flow paths that vary the most in execution time. However, the work does not attempt to identify the root-cause for this variance in execution time. The work presented in this chapter goes a step beyond what variational path profiling attempts to achieve. First, the work presented here does not rely on variations in execution time alone, but also uses information about cache misses and stall cycles in those code regions. Second, once a bottleneck code region is identified, our technique attempts to pinpoint the probable root cause for the bottleneck. Identifying the root-cause for a bottleneck has a number of advantages compared to applying a patch without knowing the root-cause for a performance bug. First, it enables programmers to apply an optimisation patch with more confidence by addressing the root of the problem (i.e., the *why* part of of the problem) and not just the symptom (i.e. the *what* part of the problem). Second, once the root-cause of a problem is identified, programmers can be confident that the optimisation patch is not introducing new bugs and may thus avoid situations like the one presented in Figure 5.1. Furthermore, our work does not make any assumptions about optimising out stall cycles for the analysis to work, rather, we utilise stall cycles in addition to execution time and cache misses like Dprof to identify the bottleneck code regions.

5.5 Threats to validity

The infrastructure proposed in this chapter monitors the runtime behaviour of a program. The idea is that monitoring actual program behaviour is perhaps more useful than speculating about the possible behaviour via static analysis. However, dynamic analysis too has its own shortcomings. First and foremost, a program may never exhibit anomalous behaviour if it is not supplied with specific bug-triggering input. The issue of generating bug-triggering program inputs has been dealt with extensively in literature. The work presented in this chapter assumes that the program is supplied with a diverse set of input or test cases, to guide the program towards anomalous behaviour. However, the assumption may not hold if either the program input set or the set of test cases are not diverse enough to trigger dormant software bugs. The approach requires at least one execution profile of the program where it's behaviour deviates [4] from the rest of the recorded execution profiles. The deviant behaviour does not necessarily need to be anomalous behaviour, i.e., it only needs to be different as compared to what has been observed in the rest of the execution profiles. If the program does not exhibit behaviour that is different than what has been observed so far, then the approach will not work and may increase

[4]An example of deviant behaviour in this case may be for a code region to consume either a significantly large or small amount of time, or a difference in the number of observed cache misses, or stall cycles in the program.

false confidence in the system.

Further, the idea behind our root-cause identification technique is based on dynamic control flow information. The analysis looks for deviant behaviour by looking for changes in the dynamic control flow path followed by the program. The idea is that if program exhibits different behaviour as compared to previous executions (such as a large amount of time/cache misses/stall cycles consumed by a code region) then we may be able to find out the root-cause by examining the control flow path followed by the program in each case. However, if the behaviour of the program does not affect the control flow path of the program, then the approach will not be able to identify a root-cause for some given bottleneck.

5.6 Chapter summary

This chapter proposed an infrastructure that is able to detect the root-cause of performance bottlenecks in parallel programs. The approach uses timing information for code regions, along with cache misses and stall cycles to identify bottleneck code regions. The idea is that rare bugs only appear under some but not all executions of the program, and under specific circumstances like a a certain program input or a specific interleaving of the program threads. The approach monitors the runtime behaviour of the program to find code regions where the program behaves suspiciously (e.g., the program spends a large amount of time in a code region in some executions of the program but not all) and logs this behaviour for offline analysis. The root-cause identification procedure makes use of program control flow information to find the root cause for the identified code regions that exhibit suspicious behaviour. The root-cause for an identified bottleneck is a point in the control flow path of the program where it deviates from the control flow path for program executions where the bottleneck does not appear. The idea is that hard-to-find bottlenecks only appear rarely and on some executions of the program with certain inputs or interleavings of the program threads. Therefore, conducting a differential analysis of the control flow path being followed by the program may shed some light on the root-cause for a bottleneck. The identified bottlenecks along with the probable root-cause for that bottleneck are then reported to developers for further inspection. The approach may not work in all the cases but has the potential to find rare performance bugs along with its probable root-cause. The chapter borrows ideas from the previous two chapters and suggests extensions to the works the work presented in previous chapters.

The suggested extensions include low overhead profiling techniques and recommendations for lightweight instrumentation frameworks. The previous approaches may improve performance if profiling data is collected by using the suggested process tracing utilities or a lightweight binary instrumentation frameworks. The use of such a lightweight framework may prove helpful in avoiding the high runtime overhead that is associated with dynamic binary instrumentation.

6 Related Work

This chapter provides a survey of previous work on improving software quality and dependability with a focus on concurrent software. The discussion focuses on both functional and non-functional concurrency bugs.

6.1 Detection techniques

Concurrency bug detection has been the focus of a number of techniques involving static, dynamic, and post-mortem analysis, as well as approaches that employ model checking. Each of the aforementioned approaches has a number of distinct properties that make them suitable for specific circumstances.

Dynamic approaches (also referred to as *on-the-fly* techniques) involve augmenting the program to detect and report data races as they occur during actual executions of the program. Dynamic techniques must maintain additional runtime information about the program state to determine when a conflicting access has occurred. Techniques based on dynamic analyses report only feasible data races (i.e., those races that have been observed during program execution and there exists a path and number of execution steps that lead to the data race). As a consequence, dynamic analysis techniques are only able to detect errors on a small number of executed paths and interleavings of the program threads, which are representative of a small portion of the whole software system. The number of feasible paths and interleavings of program threads in a program may be quite large in practice. Thus, dynamic analysis techniques may miss bugs that manifest only when a rare program path or interleaving of program threads is exercised. Further, dynamic analysis usually suffers from a high runtime overhead, and may not be suitable for programs with strict timing requirements.

Software model checking is a software verification technique that involves the algorithmic analysis of programs to prove properties of the program execution. Model checking may be viewed as very close to dynamic analysis approaches. Model checking approaches do not scale well with software size. These approaches require a specification of the software system to perform testing. Writing specifications for a large software system can be a tedious and error-prone task. Although approaches based on model checking have considerably improved, they may still run into state-space explosion problems for large scale software systems.

Post-mortem approaches analyse logs or traces of data recorded dynamically. These approaches are used in conjunction with dynamic analysis to lower the overhead incurred by online processing of the profiled data. Approaches based on post-mortem analysis are able to significantly lower the runtime overhead associated with dynamic analysis by processing of data offline. However, these approaches have to pay the cost of some space overhead for storing profiled data, to lower the runtime overhead.

Approaches that are based on static analysis are at the opposite spectrum from tools based on dynamic analysis. Static analysis does not require executing program code. Static approaches rely on dependence analysis of program instructions to determine if different references may refer to the same shared memory location. These approaches report all potential data races that could happen during program executions. Thus, static approaches may suffer from a large number of false positives, and may not work well in the presence of pointers and dynamically allocated memory.

Definition 16 (False Positives)**:** *A false positives is a potential flaw identified by program analysis that may never occur in actual program executions.*

Some approaches combine static analysis with symbolic execution to overcome some of the challenges. Symbolic execution is typically used to explore as many different paths in the program as possible in a given amount of time. For each path that is exercised, symbolic execution generates a set of concrete input values that exercise the path. Generating useful test inputs for programs that use complex operations, such as pointer manipulations and non-linear arithmetic operations, is a challenging task. Symbolic execution generally does not scale well for large programs and ends up running into the same problems as model checking.

Next, we explain a number of the aforementioned types of approaches (i.e., static, dynamic, model checking and post-mortem) in literature related to functional anomaly detection, followed by those approaches that focus on detecting non-functional anomalies.

6.2 Functional concurrency bugs

Software testing techniques for detecting functional concurrency bugs have received considerable attention in the last couple of decades. A number of effective techniques employing dynamic as well as static analysis have been proposed to detect and avoid concurrency bugs. Some notable techniques include work on instrumentation frameworks [25, 34], static inter-procedural flow analysis for detecting data races and deadlocks [32], static analysis to infer code points where synchronisation is needed to avoid data races as well as atomicity violations [129], model checking to ensure functional correctness [130], dynamic analysis for detecting the different types of concurrency bugs including atomicity violations [36, 38] and data races [96, 114], approaches for testing and exposing anomalous schedules of the program, including systematically generating feasible schedules of the program [16, 55, 139], as well as generating schedules in a randomised manner [30], and partial-order sampling techniques that attempt to uniformly exercise program schedules [116].

Most of the prior work on concurrency bug detection attempts to either achieve completeness or soundness [1]. Those approaches that focus on soundness try to suppress false warnings, i.e., to be correct in whatever is reported, and in the process they may ignore some actual bugs. As a consequence, most sound approaches may lead to false negatives, resulting in a false confidence in the system. At the opposite end of the spectrum are those approaches that report on all suspected interactions within the program and end up causing a significant number of false alarms - they may overwhelm the user with false warnings. The seminal work on the lockset approach to detecting data races introduced by Savage et al. [114] forms the basis for most of the work that chooses to focus on completeness for detecting data races in concurrent programs. Tools that aim for soundness mostly make use of techniques based on the happens-before relation. They usually end up eliminating actual defects from the list of reported bugs in their effort to be correct. Sound approaches may compromise on reporting actual defects because these approaches have limited coverage over interleavings of the program threads. As a result tools that aim for sound analysis techniques usually result in more false negatives than tools aiming for completeness which in turn suffer from false positives.

Almost all concurrency bug detection techniques suffer from either the lack of soundness or completeness. Those tools that aim for soundness and completeness [25, 38] too acknowledge missing defects hidden in certain schedules of the program. The reason is that the specific interleaving of threads that causes some bugs to surface may never happen in the observed executions of the program. Tools that aim for completeness, including Atomizer [36], Eraser [114] and most tools based on method described in [96], produce an overwhelmingly large number of false alarms.

Most of the approaches described in literature either direct their efforts to detecting a single type of concurrency bug, or have multiple procedures that must be followed to detect each specific type of bug. The different approaches to concurrency bug testing may be broadly categorised as those that focus on either data races, ordering bugs, deadlocks, or atomicity violations, and those that try to force the program into some anomalous interleaving of the program threads. These aforementioned classes of concurrency bugs are addressed in literature using either static or dynamically program analysis techniques.

The next few sections explain some of the approaches in literature that are used in detecting the aforementioned classes of concurrency bugs either statically or dynamically.

[1]Note definitions of soundness and completeness from Chapter 1, Section 1.3.1

6.2.1 Data races and ordering bugs

Research on data races and ordering bugs may be categorised as works that argue for avoiding these anomalies altogether by design, and those that work towards detecting them - either statically or dynamically.

Data race avoidance based works argue for designing programming languages in such a way that data races are impossible by definition. Some of the early work on avoiding data races includes the pioneering concept of a *monitor*, introduced by C. A. R. Hoare in 1974. A monitor is a group of shared variables along with the procedures that are allowed to access them, all bundled together and protected with a single anonymous lock that is automatically acquired and released at entry and exit of the procedures. The monitors provide a static, compile-time guarantee that accesses to shared variables are serialised and therefore, free from any data race. However, these monitors do not protect against data races in programs with dynamically allocated shared memory. Other works on data race avoidance include approaches based on compile-time analysis to avoid data races [6, 35]; however, researchers [113, 131] argue that most approaches are not viable in practice due to the prohibitively high runtime overhead associated with their technique.

Several dynamic analysis based approaches on data race detection [96, 116, 120] are inspired from Lamport's *happens-before* causality [66]. Lamport's happens-before relation is a partial order on the events of all threads in a concurrent execution. Within any single thread, events are ordered according to the same order in which they appear, whereas amongst threads they are ordered according to the properties of the synchronisation objects they access. The "happens before" relation defines the order of access to a synchronisation object between threads. An access form some thread *x* is defined to happen before another thread *y* if accesses from *x* always appear before access from *y*. The example in Figure 6.1 depicts the happens-before ordering between two different threads with respect to how they acquire the common synchronisation object. The ordering between threads is depicted by happens-before edge in the form of a solid arrow in the figure.

In the absence of any happens-before relation between threads, accesses from threads are said to happen simultaneously and could lead to opportunities for data races. Most race detectors using the happens-before relation are based on the observation that the access order of every data reference and synchronisation operation may be monitored to check for conflicting accesses to shared data. There are some significant drawbacks of approaches that employ the happens-before relation. First, the approaches using happens-before relation must record per-thread information about concurrent accesses to each shared-memory location. Recording per-thread information for every single access from all threads is a computationally intensive task and results in a high overhead. Second, the effectiveness of such approaches is highly dependent on the interleaving produced by the scheduler. A race can only be detected if the specific buggy interleaving is triggered during an observed execution of the program. For example, looking

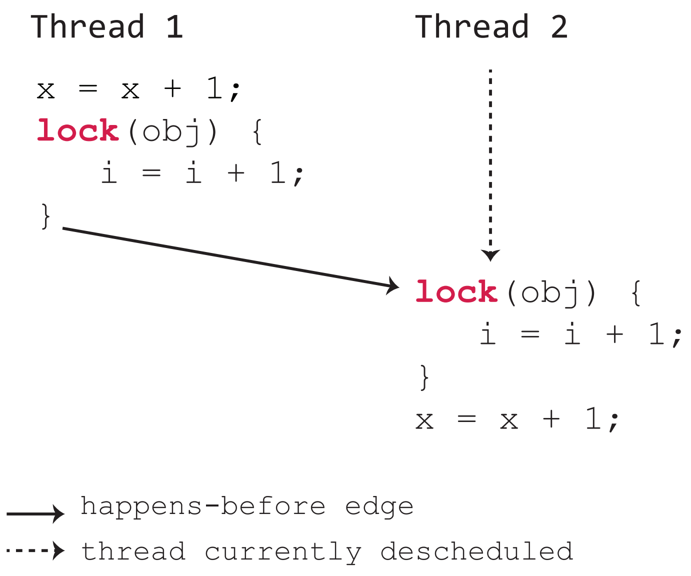


Figure 6.1: The program allows a data race on x, but the error is not detected by a *happens-before* race detector in every execution interleaving

back at the code snippet in Figure 6.1, an approach employing the happens-before relation might miss the data race in most executions of the program. The race may be detected only if the scheduler is able to produce an interleaving in which the fragment of code for *Thread2* is executed before the code for *Thread1*.

A number of dynamic approaches for data races detection, including [13, 33, 65, 96, 117, 131], are inspired from the *lockset* based approach for uncovering data races [114]. Lockset based approach improves data race detection as compared to happens-before relation based approaches. The lockset approach extends and improves upon the work by Dinning et al. [29], however, it discards the underlying happens-before apparatus used in their work. The lockset algorithm detects data races by maintaining a candidate set of locks for each shared memory location. The candidate set of locks are those locks that have protected the shared memory location for the computation performed in the program up to that point. When a shared memory location is accessed, the algorithm updates the candidate set of locks by performing an intersection of the candidate set of locks with locks being held by the current thread. If the intersection results in an empty candidate set then a warning is issued, indicating a possible data race for the shared memory location being accessed. A major problem with lockset based approaches is the significant number of false warnings about potential data races that may never happen in practice, i.e., there may be no feasible execution path that could lead to the data race.

Other dynamic techniques employ a hybrid of the happens-before and lockset based approach. O’Callahan et al. [96] have attempted a hybrid of the happens-before and lockset based approach claiming significant improvements in accuracy, fewer false positives, and reduced overhead. They argue that detection accuracy may be improved upon by applying limited happens-before checks to a lockset based approach. Accuracy may improve in the aforementioned situation because the data races reported by a full happens-before relation based tool are a subset of the races that are reported by a lockset based one. False positives are still reported, however, these are smaller in number compared to the ones reported by a lockset based tool. ThreadSanitizer [121] also employs a hybrid of the happens-before and the lockset based approach to detect data races. ThreadSanitizer introduces the use of a pre-defined set of dynamic annotations to overcome reporting false positives. A major limitation of the lockset based techniques is that these approaches check for violations of a specific locking discipline, and will fail to work if an application uses another synchronisation model or other complex locking disciplines.

While some works employ a pure static-analysis-based approach to detecting concurrency bugs [32, 88], other approaches use a combination of static and dynamic analysis. For example, the Conseq tool [148] utilises static analysis based on control and data dependencies, followed by dynamic behaviour monitoring. Conseq works by statically identify failure sites in a program binary, followed by static slicing to identify critical read instruction that may affect any of the identified failure sites. Finally, the approach monitors a dynamic execution of the program to identify any suspicious interleaving that may cause an incorrect program state.

Some more approaches for discovering data races propose hardware assisted detection. These approaches either modify the L1 cache and cache coherence protocol messages [82, 109] to include timestamps, or propose modifying the underlying hardware to include modules that would keep track of memory accesses by all processor cores to detect access conflicts [86], or works that implement the lockset based approach in hardware using bloom filters [149].

6.2.2 Inducing anomalous thread-interleaving

Traditional methods of testing concurrent programs to uncover anomalies involves repeatedly executing the program. The insight is that repeatedly executing a program may end up triggering a rare anomalous interleaving of the program threads in some executions, and expose a buggy schedule of the program. The main problem with such approaches of stress testing a software system is that rare concurrency bugs are a result of thread interleaving that may only appear under rare circumstances, such as under specific system loads or in a particular execution environment. Moreover, stress testing usually relies on the underlying operating system or virtual machine for thread scheduling. Repeated executions that do not attempt to explicitly influence thread schedules may end up executing the same interleaving of the program threads many times. There are a number of interesting techniques in literature that try to overcome the aforementioned shortcomings by influencing the scheduler in some way. Approaches that influence the scheduler employ a host of random or statistical techniques to uncover concurrency bugs.

A discussion of some notable approaches in literature that attempt to uncover functional concurrency bugs by influencing the scheduler is presented in the following text. CCI [58] is an instrumentation framework that is able to detect data races and atomicity violations. CCI tracks interleaving-related predicates (i.e., memory accesses) at runtime and uses statistical models to process the aggregated information from many executions of the program. It computes suspiciousness scores of the memory accesses, and assigns them suspiciousness scores in decreasing order. It employs random sampling to circumvent the high overhead associated with dynamic analysis. Weeratunge et al. introduce a dual slicing technique [134] that compares program schedules from execution traces. Dual slicing is a technique that compares two executions (a passing and failing one) and produces a slice that contains only those differences between the two executions that are responsible for some observably different behaviour. Dual slicing computes a set of dynamic dependencies from program executions according to a given slicing criterion that identifies some output differences between two executions. Burckhard et al. argue for a randomised scheduling algorithm [16] that is able to schedule program threads randomly for a test harness. The tool implementing the algorithm uses fine-grained priority control by implementing a user-mode scheduler. It employs binary instrumentation to insert calls to the scheduler after every instruction that accesses shared memory or makes a system call. The scheduler ensures that a thread makes progress only when all threads with higher priorities are already blocked. A problem with the scheduler is that it may starve certain threads if a high priority thread spins waiting for a lower priority one, and the program may end up in a live lock situation.

Other approaches to finding bugs by influencing thread schedules include model checking and verification, but these do not scale well for large programs. Model checkers systematically control the thread scheduler to explore all possible behaviour of a program and, hence, run into the well-known state-explosion problem. The number of possible interleavings of program threads in any mid to large-scale software system can be quite large in practice. To overcome the state-explosion problem, partial order reduction methods have been suggested [43]. The technique performs static analysis of the program to determine the interacting/dependent instructions. However, due to the limitation of static analysis in the presence of pointers and heap memory, these methods usually fail to achieve a significant reduction for real-world programs. Flanagan et al. [39] propose a partial-order reduction technique for model checking software that employs dynamic analysis. The technique is based on dynamically tracking interactions between concurrent processes/threads and then exploiting the captured information to identify backtracking points where alternative paths in the state space need to be explored. However, when using these methods in the presence of cycles in the state space, the depth of the search has to be bounded by some arbitrary bound.

6.2.3 Atomicity violations

Atomicity is a fundamental correctness properties for parallel programs. A code block is considered atomic if the program's behaviour is semantically similar to a serial execution of the atomic blocks without any interference from other threads. The behaviour of atomic blocks of code may be understood and reasoned about according to their sequential semantics, since no parallel operation can interfere with the execution of a sequentially executed block of code. A variety of dynamic analysis techniques have been proposed in literature to check for violations of atomicity property of code blocks. Checking of atomicity violations while avoiding false positives is surprisingly difficult and time consuming. As a consequence, most approaches end up reporting a large number of false positives.

A number of approaches for detecting atomicity violations have been proposed [18, 38, 39, 57, 102, 116]. Some of these approaches do fairly well in suppressing false warnings [38], however, due to the nature of dynamic analysis they may miss if a violation does not occur in the observed program executions. Atomizer [39] employs the Lockset algorithm [96], as well as ideas from Lipton's theory of reduction [73]. Atomizer suffers from some limitations including reporting false positives and being limited to synchronisation model based on mutual-exclusion locks. Further, Atomizer requires the users to manually annotate code blocks as atomic for the analysis to work, making it difficult to adapt in practice. Velodrome [38] improves upon the work of Atomizer by eliminating false positives in most cases. Velodrome uses happens-before relations for synchronisation, communication and thread ordering, and reports an error if and only if the observed trace is not conflict serialisable. An execution trace is considered conflict-serialisable if it can be transformed into an equivalent serial trace by commuting adjacent, non-conflicting operations of different threads. However, Velodrome too requires programmers to write atomicity specifications for programs to be tested beforehand. Other atomicity violation detection tools [100, 102] attempt to explore low probability interleaving of the program threads by using random sleep delays, and are able to detect violations involving a single variable only. Some more approaches [15, 18, 146] argue for detecting atomicity violations via semantic linearizability [51]. Some of these approaches attempt to numerate the many linearizations of each execution by limiting them to executions with few operations, while others require programmer annotated linearization points to reduce the exponential number of possible linearizations.

More recently, techniques have been proposed for fixing atomicity violations that involve a single variable [57, 59]. These tools work by utilising bug reports from existing concurrency bug detection tools. The techniques utilise static analysis and code transformations to analyse the code locations that are surrounding the reported concurrency bugs, and insert synchronisation constructs (protected by locks) in an attempt to eliminate the bug. The suggested patch is then applied and tested against software specifications to ensure correctness and performance. However, these techniques have a major limitation in that the fix does not provide any guarantees about program semantics. The approaches ensure that accesses are not concurrent by using mutual exclusion or pairwise ordering, however, there are no guarantees that the actual bug has been fixed and that no new bugs have been introduced. The patch-testing process usually relies on existing concurrency bug detection tools to ensure correctness.

6.2.4 Record and replay frameworks

Record and replay frameworks are an effective way of capturing dynamic properties of a program for a better understanding of its runtime behaviour. Since rare concurrency bugs are extremely hard to reproduce, record and replay frameworks offer a convenient solution. These frameworks record program execution into traces that can be replayed to mimic the exact dynamic behaviour of the program that led to its buggy behaviour. Generally, the overhead (in both space and time) associated with these frameworks is prohibitively high to be utilised in production environment, however, they serve as an excellent tool for in-house testing once a problem is suspected. Moreover, another nuisance associated with most record and replay frameworks is that they either require special execution environments or modifications to the operating system.

Record and replay frameworks have been utilised in debugging of parallel programs for more than three decades. Some of the earliest replay frameworks include [22, 69]. In the time since the early replay frameworks, researchers have come a long way and proposed many improvements to circumvent the overhead and limitations associated with these frameworks. Some of the more notable and recent record and replay frameworks include [53, 54, 74, 90, 103, 119, 133]. The detailed description of some of these aforementioned record and replay frameworks are presented in the text that follows.

Rerun [54] is a memory race detection tool and works by logging dynamic sequence of instructions that execute without interacting with other threads. The length of each such sequence of instructions is logged along with the order between sequences. It uses Lamport's clocks [66] to establish a causal order between the recorded sequences. A re-player then uses the recorded information about the sequence's duration and ordering to reconstruct an execution with the same semantic behaviour. Rerun requires modifications to the memory system architecture to work. The record and replay tool PinPlay [103] employs the dynamic binary instrumentation tool Pin [77] for both recording and replaying of programs. PinPlay consists of a logger tool that captures the initial state and non-deterministic events during program execution, and a re-player tool that uses the logged information to repeat program execution. It is able to log all inter-thread dependencies to achieve a parallel replay and guarantees to preserve the order of accesses between program threads to shared memory. PinPlay implements a software based approach, without requiring any modifications to the underlying hardware, however, the runtime overhead may be prohibitively high in some cases. A more recent deterministic record and replay tool DrDebug [133] is based on PinPlay and provides a set of record/replay tools with support for deterministic cyclic debugging for multithreaded programs in an interactive manner. Jalangi [119] is an instrumentation and dynamic analysis framework that incorporates selective record and replay as well as shadow values and shadow execution to enable writing of heavyweight dynamic analyses. Jalangi is able to deal with non-deterministic schedules and program inputs. It employs instrumentation and external proxies to record the timing of events and inputs. Some of the more recent techniques for record and replay of concurrent programs include Light [74] and rdb [53]. rdb guarantees exact replay of the observed program behaviour. The tool achieves exact replay by using hardware assisted record and replay frameworks, while keeping two versions of the application's executable. One version of the binary contains the added debug code, while the other one is kept identical to the original program binary. The runtime system of rdb then replays the application and invokes the necessary debug code at appropriate program points. Light [74] proposes a technique that guarantees determinism and as a consequence is able to reproduce concurrent bugs. Light argues that recording flow dependences is the necessary and sufficient condition to reproduce a concurrent bug. The flow dependences, together with the thread local orders that are automatically inferred, are encoded as scheduling constraints. Light attempts to reduce run-time overhead by recording only the cross-thread write-read dependences. As a result, Light requires offline constraint solving to replay an execution of the program, and is therefore, able to support only an offline replay of the program behaviour.

6.3 Performance bug detection

Software is growing in size to keep up with the increasingly complex functional as well as non-functional requirements of modern systems. The complexity of concurrent and parallel software platforms to efficiently utilise the multicore hardware of today add to the ever-growing complexity of software systems. One of the major benefit of utilising concurrent and parallel systems is that they offer higher performance throughput. Parallel software attempts to efficiently utilise the multiple cores that are available in modern systems to increase performance, however, if software is not carefully designed parallelism can have the reverse effects and may even degrade system performance. Further, the ever increasing cost of energy dictates that software systems should optimise their use of hardware more efficiently to lower energy consumption. There are a number of techniques that aid programmers in designing better software by allowing them to debug system performance at an early stage, however, performance problems are still prevalent in modern systems. Jin et al. [56] provide a comprehensive survey of performance problems in a range of deployed software systems. The findings of the aforementioned study provides a good overview of performance anomalies for a better understanding of the common characteristics of these anomalies by observing the patches that are applied to fix them. Similarly, Nistor et al. [94]

provide a study of how performance bugs are discovered, reported to developers and fixed as compared to functional bugs in popular software systems. The study argues that fixing performance bugs requires more effort as compared to fixing functional software bugs, since the former lacks better tool support.

A number of techniques have been recently proposed for dealing with performance problems in software. These include several profiling techniques that try to find asymptotic inefficiencies in software as a function of the input size [27, 45, 145] and attempt to provide hints on how to reduce the computational complexity of certain computations within a software system. Other approaches have suggested techniques for tracking computation intensive loops and proposes source code transformations to fix performance anomalies [93, 95]. Malik et al. [79] provide a case study for detecting deviations in performance based on load tests comparison. Others suggest techniques that attempt to detect performance problems as well as issues with energy consumption by generating tests that exercise anomalous program behaviour with respect to performance [17, 24, 48, 147].

Recently, several techniques have been proposed that offer automatic fixes for performance anomalies in software systems [3, 28, 67, 68, 93, 127]. These approaches utilise a wide variety of approaches to suggest automatic fixes for performance problems, including the use of genetic programming [67, 68] to offer patches for repairing particular bugs, or suggesting memiosation patches based on dynamic program anlaysis [127], static analysis offering source level fixes for loop related conditions [93], using instrumentation to collect data from failing and successful tests and using an SMT to arrive at source code level patches [28], or a differential technique that attempts to repair a target function according to the semantic difference between that target function and a reference function [3]. Most of the aforementioned approaches may not be well suited for detecting anomalies in concurrent or parallel software systems.

7 Conclusions

This dissertation presented techniques for detecting and localising functional as well as non-functional concurrency bugs. The implementation details are presented along with an experimental evaluation of the prototype implementation to demonstrate the effectiveness of these techniques. Further, the detailed description and design of an infrastructure to identify the root-causes for non-functional concurrency bugs is provided.

Multicore machines have been widely adopted in almost every walk of life. The number of CPU cores in computer systems will continue to grow in the future, leading to systems with a large number of cores performing general purpose computing. To keep up with the growing number of cores, programmers must utilise concurrency and multi-threading in their programs to fully utilise the performance potential that multicore systems have to offer. Writing multi-threaded programs is hard, and writing correct ones is doubly so. Programmers must therefore, test their programs thoroughly for the presence of bugs, however, software testing is a time-consuming and financially expensive activity. As a consequence, researcher must find better and effective techniques for testing concurrent software. The work presented in this dissertation is an effort in the same direction towards the goal of better quality software.

7.1 Summary and contributions

7.1.1 Functional bugs

The infrastructure to detect and localise functional concurrency bugs utilises dynamic control flow information about the client software to detect any deviations from expected behaviour. The infrastructure employs dynamic binary instrumentation to record the control flow path followed by client applications. The technique at the heart of the infrastructure is able to detect any deviations from the expected behaviour by tracking changes in the control flow path followed by applications. The detection and localisation of bugs works in two phases: an online phase data collection phase to record information about the control flow path followed by the program, and a post-mortem analysis phase to identify key properties of the program behaviour. The idea is that if the input to a program is fixed, and the infrastructure is able to deal with the effects of harmless non-determinism that is inherent in concurrent programs, then, the control flow path followed by a program may provide valuable information about its dynamic behaviour. The concurrency bug detection infrastructure presented in this thesis is adheres to the aforementioned idea, and is able to account for harmless non-determinism that may affect the control flow path followed by a program. Further, the infrastructure is able to localise the effects of the bug to a basic block in the program code where the bug manifest by causing a deviation in the program's control flow path.

The prototype implementation of the technique is successful in detecting and localising a wide variety of anomalies in concurrent software. The prototype is evaluated against a number of hand-crafted micro benchmarks that are representative of the types of concurrency bugs found in practice, as well as, against bugs in large scale software systems. The technique is able to detect all the bug patterns that are part of the hand-crafted micro benchmarks, as well as bugs in real-world applications including apache, aget, pbzip2, and a bug in Stringbuffer implementation of the java developer kit. Further, honouring the statement of this thesis, the approach requires only a single failed execution of the program and is able to handle the affects of harmless non-determinism that is inherent in concurrent programs. Requiring a single failed execution is important since it may not be possible to reproduce or record multiple failed executions for rare concurrency bugs. The technique is not affected by the synchronisation model followed by applications and offers a wider scope to include any bugs that may affect the control flow path followed by a program, including data races, single and multi-variable atomicity violations, and ordering bugs. The performance overhead of the approach is low and performs orders of magnitude times better than state-of-the-art approaches for concurrency bugs detection for large-scale software systems.

7.1.2 Non-functional bugs

Identifying performance problems

This dissertation presented two infrastructures for detecting non-functional concurrency bugs. First, an infrastructure is presented for detecting non-functional bugs dealing with performance problems in concurrent systems, and second, the description and design details for an infrastructure that is able to identify the root cause of performance problems is presented.

The infrastructure for identifying performance problems in concurrent systems utilises dynamic information about the execution time consumed by different parts of a applications to identify potential performance anomalies. The approach utilises profiling information about multiple executions of the program, to generate an aggregate execution profile. The generated aggregate profile is representative of the overall observed behaviour of the program and provides a better outlook on resource consumption in the program across a wide array of program inputs. Program executions are then compared against the generated aggregate execution profile to detect performance bottlenecks. The approach works by detecting rare performance bottlenecks, i.e., program behaviour that is spurious or uncommon. The infrastructure records profiling information at the level of a basic block to provide information about program behaviour at a fine-grain level. The technique does not report on code blocks that always take a large amount of execution time, because not every time-consuming code region may lead to a performance problem that can be fixed. Some code regions may involve computations that always take a large amount of time as a property of the algorithm being used and may be un-avoidable. The approach attempts to eliminate such false positives by comparing the statistical range of the maximum and minimum amount of execution time consumed by a basic block.

The prototype implementation of the technique for detecting performance anomalies is evaluated against a number of real-world applications including apache web server, a network security scanner called nmap, axel - the multithreaded download accelerator and the file compression utility pbzip. In certain cases modifying the identified basic block with little effort leads to almost three orders of magnitude improvement in application performance.

Identifying root causes of performance problems

The dissertation proposes an infrastructure that is able to identify the root-causes of performance anomalies in concurrent software. The detailed description for the proposed infrastructure is presented and the design details are laid out. The proposed framework is an extension of the works on functional and non-functional concurrency bugs presented in this dissertation. The framework combines ideas from, and improves the work presented in previous parts of this dissertation.

The proposed technique suggest an online analysis phase to profile information about program properties including timing information, cache misses and stall cycles. A number of suggestions are made for lightweight dynamic profiling of the aforementioned information to avoid a heavy performance hit at runtime. The information may be profiled at the granularity of a basic block as before, or at a slightly higher granularity by using the suggested improvements. The online profiling phase is followed by a post-mortem analysis phase of the recorded information. The post-mortem analysis consist of further two steps. The first step is concerned with identifying potential performance bottlenecks as before, but using the additional information of cache misses and stall cycles. The second step of the post-mortem analysis attempts to identify the probable root-cause for each of the identified potential bottlenecks. The root-cause identification technique boroughs ideas from the functional bug detection work that is part of this dissertation. For each identified bottleneck the root-cause identification procedure attempts to find points in the program where program execution diverges with respect to the control flow path being followed. The key insight is that if a code region consumes a large amount of time on only some executions of the program, but not all, then, executions where the code region consumes less time may contain information about this divergent behaviour of the program. The list of potential bottlenecks along with the root-cause for each are reported for inspection and may need to be verified manually by programmers.

7.2 Concluding remarks

The work of this dissertation offers an insight into some of the issues surrounding the correctness of concurrent and multi-threaded programs. The task of writing correct programs is challenging, especially in a concurrent setting. Programmers, as well as the industry in general may benefit from better tool support for ensuring correctness of their software. Current debugging support for concurrent programmes is limited. There's a greater need for better techniques and tools to ensure timely and cost-effective debugging of concurrent programs. The need for better debugging support is going to be critical as software becomes more ubiquitous, especially in safety critical environments.

This dissertation makes a small contribution to the wider, more general problem of software correctness for parallel programs. Ensuring functional, as well as, non-functional correctness of parallel program is hard but necessary with the growing need for software. This dissertation attempts to address anomalies relating to both functional and non-functional correctness of concurrent software. The work on functional correctness presented as part of this dissertation offers a technique that detects deviations from expected behaviour by observing the control flow path followed by programs. The technique is general enough to detect all concurrency bugs that cause a change in the control flow path follow by the program, however, it may miss those bugs that do not affect the program's control flow path. Similarly, the work on non-functional correctness works by examining code regions that consume a large portion of the total execution time of the program on some but not all executions of the program. The dissertation demonstrates the effectiveness of both the functional and non-functional anomaly detection techniques by offering the results of experimental evaluation of the prototype implementation of these approaches. Further, the dissertation also proposes detailed design for a framework that is able to identify the root-causes of non-functional anomalies dealing with performance of software systems.

The ideas and tools that are part of this dissertation are a small part of the effort towards accomplishing the dream of functionally correct and bug-free software systems.

Bibliography

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 74–89. ACM, 2003.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 225–236. ACM, 2014.

[4] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Production-run software failure diagnosis via hardware performance counters. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 101–112. ACM, 2013.

[5] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 307–320. USENIX Association, 2012.

[6] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of java without data races. In Mary Beth Rosson and Doug Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, pages 382–400. ACM, 2000.

[7] Enes Bajrovic and Siegfried Benkner. Automatic performance tuning of pipeline patterns for heterogeneous parallel architectures. In *The 2014 International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'14)*, July 2014.

[8] Thomas Ball. The concept of dynamic analysis. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, pages 216–234, 1999.

[9] Thomas Ball and James R. Larus. Efficient path profiling. In Stephen W. Melvin and Steve Beaty, editors, *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996*, pages 46–57. ACM/IEEE Computer Society, 1996.

[10] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In Michael B. Jones, editor, *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, May 18-21, 2003, Lihue (Kauai), Hawaii, USA*, pages 85–90. USENIX, 2003.

[11] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. PERISCOPE: an online-based distributed performance analysis tool. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 1–16. Springer, 2009.

[12] Jon L. Bentley. *Programming pearls*. Addison-Wesley, 1986.

[13] Eric Bodden and Klaus Havelund. Racer: effective race detection using aspectj. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 155–166. ACM, 2008.

[14] Randal E. Bryant and David R. O'Hallaron. *Computer systems - a programmers perspective*. Pearson Education, 2003.

[15] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 330–340. ACM, 2010.

[16] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In James C. Hoe and Vikram S. Adve, editors, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 167–178. ACM, 2010.

[17] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 463–473. IEEE, 2009.

[18] Jacob Burnim, George C. Necula, and Koushik Sen. Specifying and checking semantic atomicity for multithreaded programs. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 79–90. ACM, 2011.

[19] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1066–1071. ACM, 2011.

[20] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[21] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In William Gropp and Satoshi Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 45:1–45:12. ACM, 2013.

[22] Richard H Carver and Kuo-Chung Tai. Reproducible testing of concurrent programs based on shared variables. Master's thesis, North Carolina State University., 1985.

[23] Richard H. Carver and Kuo-Chung Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.

[24] Feifei Chen, John Grundy, Jean-Guy Schneider, Yun Yang, and Qiang He. Automated analysis of performance and energy consumption for cloud applications. In Klaus-Dieter Lange, John Murphy, Walter Binder, and José Merseguer, editors, *ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014*, pages 39–50. ACM, 2014.

[25] Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. jpredictor: a predictive runtime analysis tool for java. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 221–230. ACM, 2008.

[26] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1001–1012. ACM, 2014.

[27] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 89–98. ACM, 2012.

[28] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In Vijay Ganesh and Nicky Williams, editors, *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, Hyderabad, India, May 31, 2014*, pages 30–39. ACM, 2014.

[29] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In Barton P. Miller and Charles E. McDowell, editors, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, USA, May 20-21, 1991*, pages 85–96. ACM, 1991.

[30] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[31] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[32] Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 237–252. ACM, 2003.

[33] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 151–162. USENIX Association, 2010.

[34] Long Fei and Samuel P. Midkiff. Artemis: practical runtime monitoring of applications for execution anomalies. In Michael I. Schwartzbach and

Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 84–95. ACM, 2006.

[35] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 219–232. ACM, 2000.

[36] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71(2):89–109, 2008.

[37] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 121–133. ACM, 2009.

[38] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 293–303. ACM, 2008.

[39] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[40] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 338–349. ACM, 2003.

[41] W. Wayt Gibbs. Software's chronic crisis. *SCIENTIFIC AMERICAN*, 271(3):72, 1994.

[42] J Gilchrist. Parallel bzip2 (pbzip2) data compression software. `http://compression.ca/pbzip2`. Last accessed 05 October 2015.

[43] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[44] Patrice Godefroid and Nachiappan Nagappan. Concurrency at microsoft: An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[45] Simon Goldsmith, Alex Aiken, and Daniel Shawcross Wilkerson. Measuring empirical computational complexity. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 395–404. ACM, 2007.

[46] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In John R. White and Frances E. Allen, editors, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Massachusetts, USA, June 23-25, 1982*, pages 120–126. ACM, 1982.

[47] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd M. Austin. Demand-driven software race detection using hardware performance counters. In Ravi Iyer, Qing Yang, and Antonio González, editors, *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, pages 165–176. ACM, 2011.

[48] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 156–166. IEEE Computer Society, 2012.

[49] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5th ed.)*. Morgan Kaufmann, 2012.

[50] Kevin Henry. A crash overview of groovy. *ACM Crossroads*, 12(3):5, 2006.

[51] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[52] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, pages 117–126, 2001.

[53] Nima Honarmand and Josep Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 455–456. IEEE Computer Society, 2014.

[54] Derek Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 265–276. IEEE Computer Society, 2008.

[55] Su-Yu Hsu and Chyan-Goei Chung. A heuristic approach to path selection problem in concurrent program testing. In *Distributed Computing Systems, April 1992., Proceedings of the Third Workshop on Future Trends of*, pages 86 –92, 1992.

[56] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 77–88. ACM, 2012.

[57] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 389–400. ACM, 2011.

[58] Guoliang Jin, Aditya V. Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 241–255. ACM, 2010.

[59] Guoliang Jin, Wei Zhang, and Dongdong Deng. Automated concurrency-bug fixing. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 221–236. USENIX Association, 2012.

[60] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 110–120. ACM, 2009.

[61] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 155–170. ACM, 2011.

[62] Charles Edwin Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In Gruia-Catalin Roman and Kevin J. Sullivan, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 17–26. ACM, 2010.

[63] John C. Knight. Safety critical systems: challenges and directions. In Will Tracz, Michal Young, and Jeff Magee, editors, *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 547–550. ACM, 2002.

[64] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[65] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomás Vojnar. Healing data races on-the-fly. In Shmuel Ur and Eitan Farchi, editors, *Proceedings of the 5th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007), PADTAD 2007, London, UK, July 9, 2007*, pages 54–64. ACM, 2007.

[66] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[67] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 3–13. IEEE Computer Society, 2012.

[68] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[69] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, 36(4):471–482, 1987.

[70] Nancy G. Leveson and Clark Savage Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[71] John Levon and Philippe Elie. Oprofile: A system profiler for linux. `http://oprofile.sourceforge.net/`. Last accessed 06 July 2016.

[72] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In Ganesh

Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 609–615. Springer, 2011.

[73] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[74] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. Light: replay via tightly bounded recording. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 55–64. ACM, 2015.

[75] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1013–1024. ACM, 2014.

[76] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339. ACM, 2008.

[77] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 190–200. ACM, 2005.

[78] Gordon Lyon. Nmap–free security scanner for network exploration & security audits. `https://nmap.org/`. Last accessed 26 March 2015.

[79] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1012–1021. IEEE Computer Society, 2013.

[80] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen B. Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and François Bodin. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In *Applied*

Parallel and Scientific Computing - 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers, pages 328–342, 2012.

[81] Christoph C. Michael and Gary McGraw. Automated software test data generation for complex programs. In *The Thirteenth IEEE Conference on Automated Software Engineering, ASE 1998, Honolulu, Hawaii, USA, October 13-16, 1998*, pages 136–146. IEEE Computer Society, 1998.

[82] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In David A. Patterson, editor, *ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991.*, pages 235–244. ACM Press, 1991.

[83] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Making sense of large heaps. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 77–97. Springer, 2009.

[84] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution effciently. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 289–300. IEEE Computer Society, 2008.

[85] Ananya Muddukrishna, Peter A. Jonsson, and Mats Brorsson. Characterizing task-based OpenMP programs. *PLoS ONE*, 10(4):1–29, 04 2015.

[86] Abdullah Muzahid, Darío Suárez Gracia, Shanxiang Qi, and Josep Torrellas. Sigrace: signature-based data race detection. In Stephen W. Keckler and Luiz André Barroso, editors, *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 337–348. ACM, 2009.

[87] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[88] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 308–319. ACM, 2006.

[89] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture (ISCA 2005), 4-8 June*

2005, Madison, Wisconsin, USA, pages 284–295. IEEE Computer Society, 2005.

[90] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 22–31. ACM, 2007.

[91] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1(1):74–88, 1992.

[92] Michael Newman. Software errors cost us economy $59.5 billion annually. `http://www.abeacha.com/NIST_press_release_bugs_cost.htm`, 2002. Last accessed 06 July 2016.

[93] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. CARAMEL: detecting and fixing performance problems that have non-intrusive fixes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 902–912, 2015.

[94] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 237–246. IEEE Computer Society, 2013.

[95] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 562–571. IEEE Computer Society, 2013.

[96] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, pages 167–178, 2003.

[97] Anthony G. Oettinger. The hardware-software complementarity. *Commun. ACM*, 10(10):604–606, 1967.

[98] Manuel Oriol. The york extensible testing infrastructure (yeti). *Fundamental Approaches to Software Engineering (FASE'10)*, 2010.

[99] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Companion to the 22nd Annual ACM SIGPLAN*

Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, pages 815–816. ACM, 2007.

[100] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In Mary Jean Harrold and Gail C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 135–145. ACM, 2008.

[101] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 245–254. ACM, 2010.

[102] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In Mary Lou Soffa and Mary Jane Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 25–36. ACM, 2009.

[103] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 2–11. ACM, 2010.

[104] Erez Perelman, Trishul M. Chilimbi, and Brad Calder. Variational path profiling. In *14th International Conference on Parallel Architecture and Compilation Techniques (PACT 2005), 17-21 September 2005, St. Louis, MO, USA*, pages 7–16. IEEE Computer Society, 2005.

[105] Aleksey Pesterev, Nickolai Zeldovich, and Robert Morris. Locating cache performance bottlenecks using data profiling. In Christine Morin and Gilles Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 335–348. ACM, 2010.

[106] Sabri Pllana, Ivona Br, and Siegfried Benkner. A survey of the state of the art in performance modeling and prediction of parallel and distributed computing systems. In *Euro-Par 2004 Parallel Processing. Springer LNCS 3149:183-188*. Springer, 2004.

[107] Kevin Poulsen. Tracking the blackout bug. `http://www.securityfocus.com/news/8412`, April 2004. Last accessed 02 February 2015.

[108] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In Uwe Aßmann, Birgit Demuth, Thorsten Spitta, Georg Püschel, and Ronny Kaiser, editors, *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, volume 239 of *LNI*, page 107. GI, 2015.

[109] Milos Prvulovic. CORD: cost-effective (and nearly overhead-free) order-recording and data race detection. In *12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, February 11-15, 2006*, pages 232–243. IEEE Computer Society, 2006.

[110] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Software Eng.*, 2(4):293–300, 1976.

[111] Easwaran Raman and David I. August. Recursive data structure profiling. In Brad Calder and Benjamin G. Zorn, editors, *Proceedings of the 2005 workshop on Memory System Performance, Chicago, Illinois, USA, June 12, 2005*, pages 5–14. ACM, 2005.

[112] James Reinders. *VTune performance analyzer essentials (1st ed.* Intel Press, 2005.

[113] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In Josep Torrellas, editor, *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, San Jose, California, USA, October 21, 2006*, pages 34–41. ACM, 2006.

[114] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[115] United States Securities and Exchange Commission. SEC charges NASDAQ for failures during Facebook IPO. May 2013. Last accessed 02 March 2015.

[116] Koushik Sen. Effective random testing of concurrent programs. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 323–332. ACM, 2007.

[117] Koushik Sen. Race directed random testing of concurrent programs. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN*

2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, pages 11–21. ACM, 2008.

[118] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[119] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013.

[120] Koushik Sen, Grigore Rosu, and Gul Agha. Runtime safety analysis of multithreaded programs. In Jukka Paakki and Paola Inverardi, editors, *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, pages 337–346. ACM, 2003.

[121] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[122] Donald L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.

[123] Tianwei Sheng, Neil Vachharajani, Stéphane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. RACEZ: a lightweight and non-invasive race detection tool for production applications. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 401–410. ACM, 2011.

[124] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 561–578. ACM, 2014.

[125] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[126] Daniel Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 157–173. Springer, 2009.

[127] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: a dynamic analysis of memoization opportunities. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 607–622. ACM, 2015.

[128] User:rmT116609. Jdk-5045582 : (coll) binarysearch() fails for size larger than 1«30. `http://bugs.java.com/bugdatabase/view_bug.do?bug_id=5045582`. Last accessed 06 July 2016.

[129] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 334–345. ACM, 2006.

[130] Willem Visser and Peter C. Mehlitz. Model checking programs with java pathfinder. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, page 27, 2005.

[131] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001.*, pages 70–82, 2001.

[132] Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, and Yong Guan. Localization of concurrency bugs using shared memory access pairs. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 611–622. ACM, 2014.

[133] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In David R. Kaeli and Tipp Moseley, editors, *12th Annual*

IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014, page 98. ACM, 2014.

[134] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing concurrency bugs using dual slicing. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 253–264. ACM, 2010.

[135] Xiao Xiao, Jinguo Zhou, and Charles Zhang. Tracking data structures for postmortem analysis. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 896–899. ACM, 2011.

[136] Guoqing (Harry) Xu. Finding reusable data structures. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 1017–1034. ACM, 2012.

[137] Karim Yaghmour and Michel Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pages 13–26. USENIX, 2000.

[138] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Uncovering performance problems in java applications with reference propagation profiling. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 134–144. IEEE Computer Society, 2012.

[139] R.-D. Yang and C.-G. Chung. A path analysis approach to concurrent program testing. In *Computers and Communications, March 1990. Conference Proceedings., Ninth Annual International Phoenix Conference on*, pages 425 –432, 1990.

[140] Giridhar Appaji Nag Yas. Axel download accelerator project. `http://axel.alioth.debian.org/`. Last accessed 05 October 2015.

[141] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 159–172. ACM, 2011.

[142] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In Stephen W. Keckler and Luiz André Barroso, editors, *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 325–336. ACM, 2009.

[143] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: a device-driver case. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 193–206. ACM, 2014.

[144] S. Zaman, B. Adams, and A.E. Hassan. A qualitative study on performance bugs. In *Mining Software Repositories (MSR), June 2012 9th IEEE Working Conference on*, pages 199–208, 2012.

[145] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 67–76. ACM, 2012.

[146] Lu Zhang, Arijit Chattopadhyay, and Chao Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 4–14. IEEE, 2013.

[147] Pingyu Zhang, Sebastian G. Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 43–52. IEEE Computer Society, 2011.

[148] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas W. Reps. Conseq: detecting concurrency bugs through sequential errors. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 251–264. ACM, 2011.

[149] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: hardware-assisted lockset-based race detection. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 121–132. IEEE Computer Society, 2007.

List of Figures

List of Tables

List of Algorithms