Mälardalen University Press Licentiate Theses
No. 230

BUGS AND DEBUGGING OF CONCURRENT AND MULTICORE SOFTWARE

Sara Abbaspour Asadollah

2016

School of Innovation, Design and Engineering

ISBN 978-91-7485-261-5
ISSN 1651-9256
Printed by Arkitektkopia, Västerås, Sweden

Abstract

Multicore platforms have been widely adopted in recent years and have resulted in increased development of concurrent software. However, concurrent software is still difficult to test and debug for at least three reasons. (1) concurrency bugs involve complex **interactions among multiple threads**; (2) concurrent software have a **large interleaving space** and (3) concurrency bugs are **hard to reproduce**. Current testing techniques and solutions for concurrency bugs typically focus on exposing concurrency bugs in the large interleaving space, but they often do not provide debugging information for developers (or testers) to understand the bugs.

Debugging, the process of identifying, localizing and fixing bugs, is a key activity in software development. Debugging concurrent software is significantly more challenging than debugging sequential software mainly due to the issues like non-determinism and difficulties of reproducing failures.

This thesis investigates the first and third of the above mentioned problems in concurrent software with the aim to help developers (and testers) to better understand concurrency bugs. The thesis first identifies a number of gaps in the body of knowledge on concurrent software bugs and debugging. Second, it identifies that although a number of methods, models and tools for debugging concurrent and multicore software have already been proposed, but the body of work partially lacks a common terminology and a more recent view of the problems to solve.

Further, this thesis proposes a classification of concurrency bugs and discusses the properties of each type of bug. The thesis maps relevant studies with our proposed classification and explores concurrency-related bugs in real-world software. Specifically, it analyzes real-world concurrency bugs with respect to the severity of consequence and effort required to fix them. The thesis findings indicate that it is still hard for developers and testers to distinguish concurrency bugs from other types of software bugs. Moreover, a general con-

clusion from the investigations reveal that even if there are quite a number of studies on concurrent and multicore software debugging, there are still some issues that have not been sufficiently covered including *order violation*, *suspension* and *starvation*.

To my beloved Family

&

To whom it may read

Acknowledgment

My most earnest acknowledgment must go to my supervisor, Prof. Hans Hansson, for his extraordinary guidance, caring, and patience. As an excellent supervisor and researcher, he will be a great example throughout my professional life.

This thesis would not exist without the contributions of my co-supervisors Prof. Daniel Sundmark and Dr. Sigrid Eldh for their continuous effort to support and encourage me. Their invaluable suggestions and discussions played an important role in improving this thesis. Thank you!

I am very grateful to my colleagues and friends, Dr. Rafia Inam, Dr. Wasif Afzal and Eduard Paul Enoiu for their supports, discussions and feedbacks as co-authors in my published papers. Also thanks to Prof. Elaine Weyuker and Prof. Thomas Ostrand for useful discussions.

I would also like to thank all my friends and colleagues at Mälardalen University providing a fruitful environment and giving support when I have needed.

From the bottom of my heart, I would like to extend my deepest gratitude to my parents as well as my brother and sister for their unconditional support, love, and faith in many phases of my life. Without their support I would not have been able to reach here.

Above all, I thank God for helping me and sending people who have been such strong influence in my life and giving confidence at my hard moments. Thank You for always being there to bless and guide me.

This research has been supported by Swedish Foundation for Strategic Research (SSF) via the SYNOPSIS project.

Sara Abbaspour Asadollah
Västerås, March 21, 2016

List of publications

Papers included in the licentiate thesis[1]

Paper A *Towards Classification of Concurrency Bugs Based on Observable Properties*, Sara Abbaspour Asadollah, Hans Hansson, Daniel Sundmark, Sigrid Eldh. In the Proceedings of the 1st International Workshop on Complex faults and failures in large software systems (COUFLESS), ICSE 2015 Workshop, May 2015.

Paper B *10 Years of Research on Debugging Concurrent and Multicore Software: A Systematic Mapping Study*, Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson and Wasif Afzal. Software Quality Journal, January 2016.

Paper C *A Study of Concurrency Bugs in an Open Source Software*, Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson and Eduard Paul Enoiu. In the proceedings of the 12th International Conference on Open Source Systems (OSS), May 2016.

Additional papers, not included in the licentiate thesis

A Survey on Testing for Cyber Physical System, Sara Abbaspour Asadollah, Rafia Inam, Hans Hansson. In the Proceedings of the 27th International Conference on Testing Software and Systems (ICTSS), Lecture Notes in Computer Science series, November 2015.

[1] The included articles have been reformatted to comply with the licentiate thesis layout.

Contents

I

Thesis

Chapter 1

Introduction

In the mid 1980s companies manufactured versions of some single core processors with two cores on one chip (dual core). Later, in the early 2000s, the manufacturing changed by Intel, AMD, IBM and other companies to development of more pure multicore processors. There is an ongoing change in hardware to improve systems' performance by increasing the number of cores. Hardware providers such as Intel and IBM, steadily increase the number of processor cores. In the past few decades, the performance of processors has been continuously increasing at exponential rates [1]. Due to the changes, there are constantly new demands to adapt to the latest execution paradigm provided by parallelism. Multicore platforms have resulted in an increase in development of concurrent software. Today, in 2016, many types of computing systems, from desktops and mobile systems to Internet cloud systems and cyber-physical systems, are dependent on multicore platforms.

From a software developer point of view, concurrent software introduces the possibility of new types of software bugs, known as concurrency bugs [2]. Concurrent software may exhibit problems, like deadlocks and race conditions that may not occur in sequential software. The errors typically appear under very specific (nondeterministic) thread interleavings between shared memory accesses. The effects of the bugs spread through the software until they cause the software to hang, crash or produce incorrect output. Such nondeterministic bugs are typically considered to be problematic errors [3, 4, 5].

Concurrency bugs in deployed systems can result in serious disasters. For instance, in 2003, ten million people were out of power due to a race condition in a monitoring software with multi-million lines of code (the often cited 2003

Northeastern U.S. electricity blackout [6]). Facebook's initial public offering (IPO) was delayed by more than half an hour, leading to a loss of millions of dollars due to a race condition in NASDAQ's IT systems [7]. It is extremely important for businesses to avoid these catastrophic losses. In 2007 a survey was conducted by Microsoft researchers to assess the state of the practice of concurrency in their products. The research indicated that over 60% of respondents had to deal with concurrency issues and half of the concurrency issues occurred at least monthly [4].

Debugging is a separate process and a key activity in software development. It involves several steps i.e., identifying, localizing and fixing bugs. One step in the testing and debugging process is determining a problem (bug or fault) in software. This phase is frequently called *bug or fault identification*. To be able to determine the problem, often the bug is replicated and information gathered. At some point, the bug will reach a developer (or tester), and it is often here the actual debugging process starts. The next step is identifying the right part of a software component, typically a smaller part of the software, e.g. an identity like file (or files) that are involved in the bug. This phase is frequently called *bug or fault localization*. The bugs and their location must be found [8] before the root cause can be identified. At this point we assume that the developer have at least pinpointed the files, code sections and general location of the bug, by utilizing e.g. minimization techniques [9] and been able to reproduce the bug in context. The final step of the debugging process is repairing and fixing the bug in order to remove it from the software.

Most experimental studies on concurrent and multicore software provide information on application cost, efficiency and complementary aspects of the testing criteria, while there is still lack of knowledge on debugging criteria evaluation to support the prevention and detection of bugs. It is thus important to have deepen the knowledge on evaluation of debugging criteria and fixing concurrency bugs.

1.1 Concurrent Software Challenges

Concurrent software test and debug compared to corresponding activities for sequential software is faced with a variety of challenges. The main challenges are as follows:

- Concurrency bugs typically involve changes in program state due to particular interleavings of multiple threads of execution, which can make them difficult to find and understand. Therefor, many concurrency bugs

remain hidden in programs (or source code) until the software runs in a real environment, and even then it may take a long time before the bug manifests itself.

- The thread interleavings may vary widely dependent on the platform selected for software execution. The platform could be a single-core or a multicore. The different run-time thread interleavings (scenarios) need to be thoroughly considered and handled to guarantee predictability in a wide range of environments. Therefor, the type of run-time environment which is selected for software execution is an important consideration.
- Repeated execution of the same concurrent source code will typically not guarantee the same result after each execution. In other words, if there are different interleavings of thread executions, then different outputs may be obtained. Consequently, developers might not be able to systematically reproduce the bug using traditional debugging methods. In general, reproducing the thread schedule, which led developers to the same bug, might be very difficult. Thus, nondeterministic thread scenarios make concurrent software test and debug extremely difficult.

1.2 Motivation and Goal of Thesis

This research is carried out in the context of concurrent software debugging. It outlines the issues involved in debugging software on concurrent and multicore architectures. Three goals are considered in this thesis:

- **Goal 1**: To provide a common terminology for distinguishing between different types and classes of concurrency bugs and to identify the interrelation between separate elements and classes.
- **Goal 2**: To identify the current gaps and less-explored areas in debugging of concurrency bugs.
- **Goal 3**: To identify the current state of concurrency related bugs in real-world software in terms of frequency, severity and resolving time.

1.3 Research Method

The methodology that has been used in the research consists of three main study methodologies. We started to generate a theory by presenting a classi-

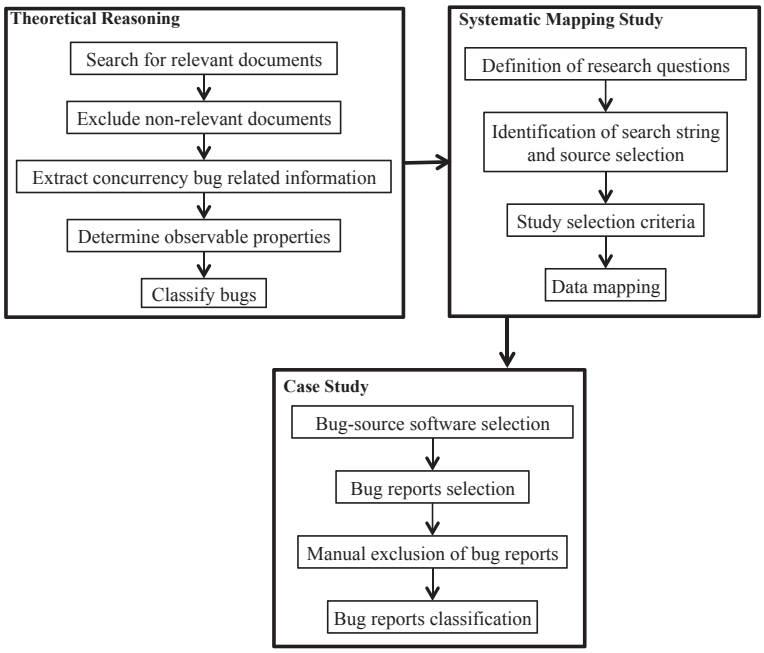


Figure 1.1: Research method

fication of bugs related to concurrent execution of application level software threads. Then, we performed a systematic mapping study for each published article by identifying the type of bug(s) and the addressed phase(s) in the debugging process. Finally, we explored the nature and extent of concurrency bugs in real-world software by performing a case study. Figure 1.1 shows the research method process.

The details are summarized as follows:

- *Theoretical reasoning* by performing a **grounded theory study**. A grounded theory study seeks to generate a theory which relates to the particular situation forming the focus of the study [10].

- *Systematic mapping study* by performing a **systematic literature review**. A systematic literature review is a formalized, repeatable process in which researchers systematically search a body of literature

to document the state of knowledge on a particular subject.

- *Case study* by performing a **case study** on the bug reports from an open source software project. Case study is a flexible empirical method used for primarily exploratory investigations that attempt to understand and explain phenomenon or construct a theory [11].

1.4 Research Contribution

The separation and identification of concurrency bugs and non-concurrency bugs is considered in order to fulfill the research goals by studying the properties of different types of concurrency bugs. The differences between concurrency and non-concurrency bugs is examined in terms of frequency, severity and fixing time. In addition, concurrency bug types are compared.

The results are disseminated in a journal article, a conference and a workshop paper. The following sub-sections briefly present explanations of each paper and Table 1.1 shows the contributions of the individual papers and their relative research goals.

To achieve Goal 1 we proposed a disjoint classification for concurrency bugs by classifying the bugs in a common structure considering relevant observable properties.

We provided an overview of existing research on concurrent and multicore software debugging. We applied the systematic mapping study method in order to summarize the recent publication trends and clarify current research gaps in the field. Based on the obtained results we summarized the publication trend in the field during the last decade by showing distributions of publications with respect to *year*, *publication venues*, *representation of academia and industry*, and *active research institutes*. We also identified research gaps in the field based on attributes such as *types of concurrency bugs*, *types of debugging processes*, *types of research* and *research contributions*. The results of our mapping study also indicate that the current body of knowledge concerning debugging concurrent and multicore software does not report studies on many of the other types of bugs or on the debugging process. In other words, there are still quite a number of issues and aspects that have not been sufficiently covered in the field. By that we address Goal 2.

Moreover, we investigated the bug reports from an open source software project (Apache Hadoop). Hadoop has changed constantly, 59 releases, over

six years of development. It has an issue management platform for managing, configuring and testing. Our results indicate that a relatively small share of bugs is related to concurrency issues, while the vast majority are non-concurrency bugs. Fixing time for concurrency and non-concurrency bugs is different but this difference is relatively small. In addition, concurrency bugs are considered to be slightly more severe than non-concurrency bugs. By this we address Goal 3.

More details about the research results are presented in Chapter 4.

Table 1.1: The contribution of the individual papers to the research goals

Papers	Goal 1	Goal 2	Goal 3
Paper A	✓	✓	✓
Paper B		✓	
Paper C			✓

1.4.1 Publications Included in the Thesis

Paper A

Towards Classification of Concurrency Bugs Based on Observable Properties [12]

Sara Abbaspour Asadollah, Hans Hansson, Daniel Sundmark, Sigrid Eldh
Status: Published in the Proceedings of the 1st International Workshop on Complex faults and failures in large software systems (COUFLESS), ICSE 2015 Workshop, IEEE, May 2015.

Abstract In software engineering, classification is a way to find an organized structure of knowledge about objects. Classification serves to investigate the relationship between the items to be classified, and can be used to identify the current gaps in the field. In many cases users are able to order and relate objects by fitting them in a category. This paper presents initial work on a taxonomy for classification of errors (bugs) related to concurrent execution of application level software threads. By classifying concurrency bugs based on their corresponding observable properties, this research aims to examine and structure the state of the art in this field, as well as to provide practitioner support for testing and debugging of concurrent software. We also show how the proposed classification, and the different classes of bugs, relates

to the state of the art in the field by providing a mapping of the classification to a number of recently published papers in the software engineering field.

Personal contribution: I am the initiator, main driver and author of all parts in this paper. All other co-authors have contributed with valuable discussion and reviews.

Paper B

10 Years of Research on Debugging Concurrent and Multicore Software: A Systematic Mapping Study [13]

Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson and Wasif Afzal
Status: Published in the Software Quality Journal, January 2016.

Abstract Debugging – the process of identifying, localizing and fixing bugs – is a key activity in software development. Due to issues such as non-determinism and difficulties of reproducing failures, debugging concurrent software is significantly more challenging than debugging sequential software. A number of methods, models and tools for debugging concurrent and multicore software have been proposed, but the body of work partially lacks a common terminology and a more recent view of the problems to solve. This suggests the need for a classification, and an up-to-date comprehensive overview of the area.

This paper presents the results of a systematic mapping study in the field of debugging of concurrent and multicore software in the last decade (2005–2014). The study is guided by two objectives: (1) to summarize the recent publication trends and (2) to clarify current research gaps in the field.

Through a multi-stage selection process, we identified 145 relevant papers. Based on these, we summarize the publication trend in the field by showing distribution of publications with respect to *year*, *publication venues*, *representation of academia and industry*, and *active research institutes*. We also identify research gaps in the field based on attributes such as *types of concurrency bugs*, *types of debugging processes*, *types of research* and *research contributions*.

The main observations from the study are that during the years 2005–2014: (1) there is no focal conference or venue to publish papers in this area, hence a large variety of conferences and journal venues (90) are used to publish rele-

vant papers in this area; (2) in terms of publication contribution, academia was more active in this area than industry; (3) most publications in the field address the data race bug; (4) bug identification is the most common stage of debugging addressed by articles in the period; (5) there are six types of research approaches found, with solution proposals being the most common one; and (6) the published papers essentially focus on four different types of contributions, with ”methods” being the type most common one.

We can further conclude that there is still quite a number of aspects that are not sufficiently covered in the field, most notably including (1) *exploring correction* and *fixing bugs* in terms of debugging process; (2) *order violation*, *suspension* and *starvation* in terms of concurrency bugs; (3) *validation* and *evaluation research* in the matter of research type; (4) *metric* in terms of research contribution. It is clear that the concurrent, parallel and multicore software community needs broader studies in debugging.This systematic mapping study can help direct such efforts.

Personal contribution: I am the main driver and author of this paper. All other co-authors have contributed with valuable discussion useful idea and reviews.

Paper C

A Study on Concurrency Bugs in an Open Source Software [14]

Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson and Eduard Paul Enoiu

Status: Published in the proceedings of the 12th International Conference on Open Source Systems (OSS), May 2016.

Abstract Concurrent programming puts demands on software debugging and testing, as concurrent software may exhibit problems not present in sequential software, e.g., deadlocks and race conditions. In aiming to increase efficiency and effectiveness of debugging and bug-fixing for concurrent software, a deep understanding of concurrency bugs, their frequency and fixing-times would be helpful. Similarly, to design effective tools and techniques for testing and debugging concurrent software understanding the differences between non-concurrency and concurrency bugs in real-word software would be useful.

This paper presents an empirical study focusing on understanding the differences and similarities between concurrency bugs and other bugs, as well as the differences among various concurrency bug types in terms of their severity and their fixing time. Our basis is a comprehensive analysis of bug reports covering several generations of an open source software system. The analysis involves a total of 4872 bug reports from the last decade, including 221 reports related to concurrency bugs. We found that concurrency bugs are different from other bugs in terms of their fixing time and their severity. Our findings shed light on concurrency bugs and could thereby influence future design and development of concurrent software, their debugging and testing, as well as related tools.

Personal contribution: I am the main driver and author of all parts in this paper. My supervisors contributed with valuable discussion, useful idea and review of the whole paper. Eduard Paul Enoiu contributed by valuable discussion, reviewing and proofreading of Section 8.4.

1.5 Outline of the Thesis

This thesis is organized in 8 chapters. Chapter 2 introduces the required background of the thesis. In Chapter 3 we present a cross-section of related work relevant to this thesis. Chapter 4 presents the results according to the respective research goals, introduced in Section 1.2. Finally, in Chapter 5 we present a discussion based on our obtained results, a list of conclusions from development of this thesis as well as possible future work, followed by the included papers in Chapter 6 to 8.

Chapter 2

Background

In this chapter we provide background information needed for understanding the context of the thesis and the work itself.

2.1 System Architecture

There are two main trends in multicore architecture systems: Symmetric Multiprocessing (SMP) and Asymmetric Multiprocessing (AMP). In SMP, all CPU cores are identical. If a programmer writes a code to run on one core then the code can run on any of the SMP cores. In AMP, different CPU cores can have different roles with different kernels running on different cores. In this thesis, our focus is on SMP type architectures. The reason for focusing on SMPs is that the memory and I/O devices are shared equally among all of the processors in the system [15]. They are more uniform and we believe that concurrency problems appear in a more similar way among SMPs than AMPs, which implies that articles relaying to concurrency in SMPs are straightforward to classify. Typically, SMP systems scale from one processor to as many as 36 processors [15]. Figure 2.1 shows the architecture model of the SMP system. In this SMP model the system have a single-chip multicore processor with "k"' identical cores and two levels of cache[1]. Each core has its private level one cache, while the last level cache (LLC) is shared among all cores. We furthermore assume a single operating system managing resources and execution on all cores.

[1] Cache is "an area of memory that holds recent used data and instruction" [16].

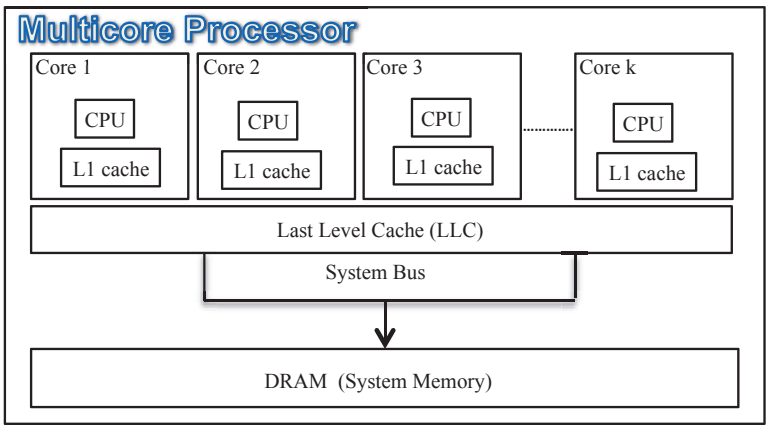


Figure 2.1: System hardware architecture

The scheduler is responsible for scheduling multiple threads simultaneously on all cores. It initiates the multi-threaded program on one core and instructs each core to start processing. As shown in Figure 2.2 we assume that there is a global (single) ready queue and a single waiting queue for each (non-CPU) shared resource in the system. The queues are shared among all cores. The scheduler uses different resource sharing protocols to synchronize the multi-threaded program. When multiple threads attempt to access a shared resource or a critical section (that is protected by a synchronization protocol), only one thread at a time is allowed to access the resource. All other threads will wait until the resource becomes free.

Migrating code from a single core environment to an SMP multicore may give rise to the occurrence of new bugs due to the concurrent execution of tasks (e.g. related to data races) that cannot occur when only one thread executes at a time in a single-core environment. The traditional single-core resource sharing protocols may not be completely helpful in eradicating these newly generated bugs.

2.2 Debugging Techniques

Debugging is a key activity in the software development life-cycle. Debugging is a methodical process of identifying, localizing, reducing and fixing bugs in

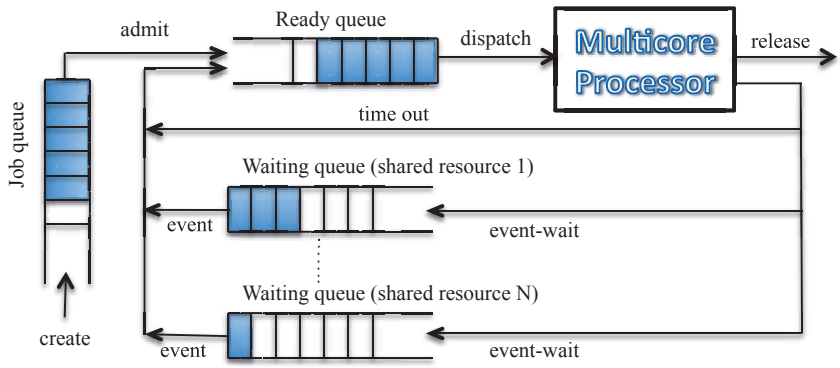


Figure 2.2: Scheduling queues

a computer program. There are a number of tricks (methods) that can be used in the daily software development activity to facilitate the hunt for software problems (bugs). Some of these methods are as follows:

- ***Exploiting compiler features***: programmers can obtain static analysis of the code provided e.g. by the compiler. Static code analysis is the analysis of software that is performed without actual executing it. Such analysis helps programmers detect a number of basic semantic problems, e.g. type mismatch or dead code.

- ***Abused cout debugging***: the cout technique[2] consists of adding print statements in the code to track the control flow and data values during code execution (also known as Print debugging or Echo Debugging). This technique is the favorite technique of beginners and has been the most common method for debugging [17].

- ***Logging***: logging is another common technique for debugging. This technique automatically record information messages or events to monitor the status of the program in order to diagnose problems.

- ***Assertions and defensive programming***: assertions are expressions, which should evaluate to true at a specific point in the code. If an assertion fails, a bug is found. The bug could possibly be in the

[2]cout technique's name is taken from the C++ statement for printing on terminal screen (or any standard output stream).

assertion, but more likely it will be in the code. In this method after an assertion fails it makes no sense to re-execute the program.

- ***Debugger***: a debugger works through the code line-by-line in order to make the execution visible to the developer, thereby helping to find bugs, the location of bugs and the cause of bugs. It can work interactively by controlling the execution of the program and stopping it at various times, inspecting variables, changing code flow whilst running, etc. Trace debugging, Omniscient debugging techniques [17] and Deterministic Replay Debugging (DRD) [18] can be considered as subgroups of this technique.

In addition to traditional debugging techniques, concurrent and parallel programs have specific debugging techniques to support tracing and debugging multithreaded software. These techniques include:

- ***Event-based debugging***: regards the execution of parallel programs as a series of events and records and analyzes the events in debugging when a program is executing. Instant Replay [19] can be considered as a type of this group.

- ***Control information analysis***: this technique can analyze the control information in execution and the global data.

- ***Data-flow-based static analysis***: this technique can detect and analyze the bugs when a program does not execute.

2.3 Types of Concurrency Bugs

Concurrent programming puts demands on software development and testing. Concurrent software may exhibit problems that may not occur in sequential software. There is a variety of challenges related to faults and errors in concurrent, multicore and multi-threaded applications [20, 21, 22]. One of the well-known concurrency bugs is *Data race*. *Data race* requires that at least two threads access the same data and at least one of them write the data [23]. It occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [20] [24]. Figure 2.3 shows an example of a *Data race*.

The following sequential actions will happen in executing the indicated code in each thread in the example:

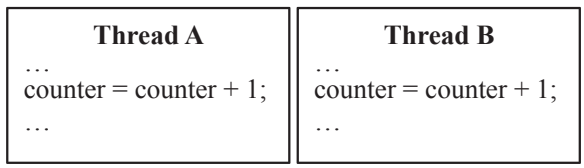


Figure 2.3: Data race example

1. Load the value of counter in memory.
2. Add 1 to the value.
3. Save the new value to counter.

Consider that this example is a small part of an application which is executing on the SMP architecture explained in Section 2.1. Suppose that threads A and B execute in parallel on Core1 and Core2 and that the value of *counter* is 100 initially. After execution, the value of *counter* could be 101 while the expected (correct) result is 102. Both cores execute the indicated line of code, but due to the parallel execution the second load is in this scenario performed before the first save. Hence, the value saved by both threads will be 101. This scenario shows that the result of parallel execution of the example could be incorrect. Thus a concurrency bug (*Data dace*) has happened.

Atomicity violation is another type of concurrency bug. It refers to the situation when the execution of two code blocks (sequences of statements) in one thread is concurrently overlapping with the execution of one or more code blocks of other threads in such a way that the result is not consistent with any execution where the blocks of the first thread are executed without being overlapping with any other code block. Figure 2.4 shows an example of single variable atomicity, and Table 2.1 displays the values of shared and local variables after each interleaving execution.

Suppose Thread A is executing on Core1 and Thread B on Core2. Both of them use a shared variable *counter* and each has its local variable (*tempA* and *tempB*). The initial value of *counter* is 0. Since both threads are using the lock mechanism to protect from data corruption, only one core at a time can access the *counter*. If Core1 reaches line 5 before Core2 reaches line 17 then the *counter* will be fetched from DRAM to LLC and L1 Cache of Core1. *tempA* will be fetched similarly. The value of *tempA* will be 0 after executing line 6 and 7. Meanwhile if Core2 reaches line 17 then Thread B will wait in the waiting queue. By releasing the lock by Core1 Thread B will wait in ready queue. Since Core2 is free and no more threads is waiting in ready queue then

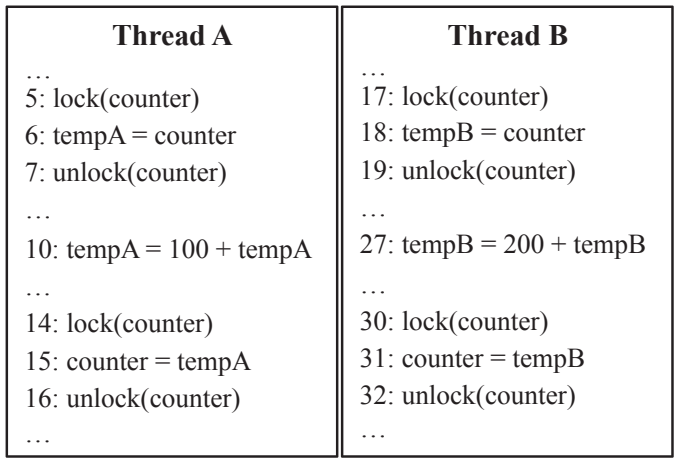


Figure 2.4: Atomicity violation example

	L1 cache of Core1		L1 cache of Core2		LLC			DRAM		
	counter	tempA	counter	tempB	counter	Core1 tempA	Core2 tempB	counter	Core1 tempA	Core2 tempB
1	0	0			0	0	0	0	0	0
2				0	0	0	0	0	0	0
3	0	100	0	200	0	100	200	0	100	200
4	100	100			100	100	1000	100	100	1000
5			200	200	200	100	200	200	100	200

Table 2.1: Shared and local variables' value after interleaving execution

Core2 will continue to execute Thread B from line 17, 18 and 19. The value of *counter* will be fetched to L1 Cache of Core2 and the *tempB* value of Thread B will be 0. During Core2 execution Core1 is executing Thread A. The *tempA* value of Thread A will be 100 while the *tempB* value of Thread B becomes 200. If we suppose Core1 reaches line 14 before Core2 reaches line 30 then 100 will be stored in LLC and DRAM as *counter* value, and then Core2 will continue (line 30, 31 and 32) and store 200 in LLC and DRAM. This scenario shows that a concurrency bug (*Single variable atomicity violation*) occurred because the updated *counter* by Core1 is corrupted by Core2. From the above

examples it should be clear that concurrent executions of threads may lead to bugs that are not possible when executing the same threads on a single core architecture. Investigating and understanding such bugs is the main motivation and focus of this thesis.

2.4 Debugging Process

In this section, we present the concepts of the different phases in the debugging process. We discuss the stages that follow after a software failure has been observed, when its root cause is determined and corrected.

From an industrial perspective, a simple life cycle of a software problem is defined by Zeller [9] to include the following phases: (1) A user reports a problem to the software provider; (2) A developer at the software provider reproduces the problem; (3) The developer isolates the circumstances of the problem; (4) The developer fixes the problem locally; (5) The developer delivers the fix(es) to the user.

The debugging process is handled differently in different types of organizations and teams. In a small team with few developers, it is normally clear what part of the code is in question when a program executes unsuccessfully or a test case fails. Here, typically, the developer has to find the bug [25]. In larger organizations, usually the first sign of any bug is the failure of the software or system. The bug fixing process then starts with the submission of an anomaly report. The following list discusses the stages that follow after a software failure has been observed, and its root cause should be determined and corrected.

- ***Bug identification*** is the process of finding the approximate location of a bug (in terms of source code unit, sub-system or even organizational unit), such that the remainder of the debugging process can be assigned to the appropriate stakeholder. It is to be noted that the scope of our definition of bug identification covers terms such as bug localization and bug detection.

  In case the failure was detected during testing, bug identification is usually performed by the testing team and followed by a team review to prioritize fixes [26].

- ***Type of bug identification*** is a process to help developers in finding the real cause of a bug by understanding the type of bug. In [12] we

extended the common debugging process by adding a sub-process that suggests that before the type of bug is identified, developers could check the properties of identified bug(s) and compare them with the properties given for each class of concurrency bugs. Thus, developer(s) can thereby identify the potential type of the bug at hand.

- In ***cause identification***, the root cause of a bug is identified. Since the root cause refers to the most basic reason(s) for the occurrence of a bug, during this process a bug can reasonably be identified by a developer or the debugger (e.g., unexpected value of variable A was the root cause of a bug related to variable B or an erroneous lock was the root cause of bug number 5).

- The process of ***exploring corrections*** can be applicable when we have more than one possible solution for fixing the bug. Typically the potential solutions are compared and the best solution for the current bug is selected.

- Finally, ***fixing bug*** is the process for repairing and fixing the current bugs. It is the last stage of the debugging process in order to remove the bug.

Note that, after debugging is competed the fixed system needs to be tested to endure that the fix did not introduce new bugs in the system.

Chapter 3

Related Work

This chapter presents a cross-section of related work relevant to this thesis.

3.1 Empirical Studies on Concurrent Software

There are some empirical studies investigating how programmers develop concurrent software [27, 28, 29, 30, 31, 32]. These studies evaluate several aspects of the work of beginners and experienced developers, such as how they design programs, how much speedup they achieve by their design, how concisely they write programs. However, the studies do not evaluate how much time developers need to fix a concurrency bug or how developers debug concurrency bugs. We, on the other hand, investigate these issues as a part of our study. To our knowledge, there are only two related studies on debugging concurrent programs. The first of these is done by Lönnberg et al. to investigate how students understand concurrency bugs [33]. The authors performed an empirical study on students, by providing an assignment to students (to write concurrent programs). They suggested several ways to help students debug their assignments. For instance, they guided students to use software visualization tools. Further, the authors interviewed the students and analyzed their responses. The authors claim that since students usually have different understanding of concurrent programs from teachers, software visualization tools will help both teachers and students to get the same view of the programs and bugs. The second study is done by Sadowski and Yi to show how developers use a new concurrency notation called *cooperability* [34]. They posted three concurrency bugs on an

internet-based survey form, divided participants into two groups, where one group of people have the aid of *cooperability* and the others do not. In evaluating the responses they scored the correctness of the responses with a ranking scheme and statistically showed that developers can understand concurrency bugs better with the aid of *cooperability*.

There are also related studies on concurrency bug types, detecting a type of concurrency bugs and reproducibility of bugs. Lu et al. examined concurrency bug patterns, manifestation, and fix strategies of 105 randomly selected real-world concurrency bugs from four open-source application (MySQL, Apache, Mozilla and OpenOffice) bug databases [35]. Their study focused on several aspects of the causes of concurrency bugs, and the study of their effects was limited to determining whether they caused deadlocks or not. We use a similar study methodology in our case study to find relevant bug reports for our analysis, but we provide a complementary angle by studying the effects of recent concurrency bugs with a more fine-grained classification than mapping bugs in to deadlock and non-deadlock bug classes.

The study by Gu et al. [36] look at the change history for thread synchronization. The authors investigate code repositories of open-source multi-threaded software projects to understand synchronization challenges encountered by real-world developers. They reviewed over 250,000 revisions of four representative open source software projects to distinguish how developers handle synchronizations. Further, the authors conduct case studies to better understand how concurrency bugs are introduced by code changes and how developers handle synchronization problems. Gu et al. conclude that it is necessary to have tool support to help developers who tackle synchronization problems.

Schimmel et al. [37] present an empirical evaluation of bug detection capabilities of two data race bug detection tools on real-world concurrent software. The authors tracked 25 data races in bug repositories, created parallel unit tests and executed 4 different data race detectors. They conclude that with a combination of all detectors 92% of the contained data races can be found, whereas the best data race detector only finds about 50%.

The reproducibility of bugs is analyzed in [38]. The authors distinguish concurrency bugs from non-concurrency bugs when trying to characterize their reproducibility. The study analyzes some applications focusing on the properties of the inputs that are required to trigger bugs. The main focus is not on concurrency bugs.

3.2 Tools for Debugging Concurrent Software

In order to help developers to debug concurrent software and trace the thread interactions some visualization tools such as CHESS [39], JPF [40], TIE [41], JIVE [42, 43], JOVE [42, 43], FALCON [44], UNICORN [45], GRIFFIN [46] and Concurrency Explorer [39] are proposed. Most of these tools are evaluated with toy programs and not with real concurrent software, except the Concurrency Explorer, which is used internally at Microsoft.

In addition, there are some tools proposed by researchers for detecting concurrency bugs, including data race detectors, serializability violation detectors, atomicity violation detectors and other bug detectors. Data race detectors can typically be of three different types based on the algorithms that are used. The first type relies on the lockset algorithm [47] to check whether the software developer protected all accesses to a specific shared variable with a common lock. The second type relies on the happens-before algorithm [48, 49] and the third type relies on sampling and the use of breakpoints [50] instead of relying on any of these algorithms. Typically, race detectors operate at the lower-level of individual memory accesses. However, Artho et al. [51] investigate data races on a higher abstraction layer. The authors developed a runtime analysis algorithm to detect high-level data races. They introduce a concept of view consistency and utilize it to detect high-level data races. A view is the entire set of shared variables accessed in a synchronized block. According to the authors, by their algorithms they can detect inconsistent uses of shared variables, even if no classical race condition occurs.

Xu et al. [52] propose a serializability violation detector to detect erroneous executions of shared-memory programs without requiring a priori program annotations. Their tool can report some dynamic false positives, which makes it particularly suitable to be used in avoiding erroneous executions caused by unknown bugs. The authors validate their proposed method by conducting an empirical case study and claim that the experimental results show that the method is effective on real server programs.

Lu et al. propose a tool that detects atomicity violation at the level of individual memory accesses (low-level) [53]. It relies on training and can detect atomicity violation bugs by learning from a large set of runs of valid memory access patterns.

A bug detector tool is proposed by Huang et al. [54]. Their tool relies on detecting whether critical sections are commutative. The authors achieve this by identifying pairs of critical sections that non-deterministically change the contents of shared memory due to execution order.

Other researchers have addressed the problem of detecting concurrency bugs in different types of event-based frameworks [55, 56, 57]. In our study we present and classify relevant papers that propose concurrency debugging tool(s).

3.3 Literature Reviews and Classification Studies on Concurrent Software

There are some SLR, surveys and state of art review studies related to concurrent software testing and debugging. These studies provide a list of relevant studies in the area. A systematic review on concurrent software testing was published by Brito et al. [58] in 2010. Their main goal was to obtain evidence of current state-of-the-art related to testing criteria, testing tools and to find bug taxonomies for concurrent and parallel programs. They further provided a list of relevant studies as a foundation for new research in the area. The authors concluded that there is a lack of testing criteria and tools for concurrent programs. They notice that most experimental studies are providing information on application cost, efficiency and complementary aspects, while there is lack of knowledge on bug taxonomy and on evaluating testing criteria. We use a similar study methodology (Systematic Mapping Study) with focus on current state of research related to debugging criteria rather than testing. However, our study is based on different classifications compared to Brito et al.'s study.

A state of the art review on deterministic replay debugging in multithread programming was performed by Wang et al. [59] in 2012. They categorize replay-based debugging techniques for parallel and multithread programs and divided them into three types: hardware-based, software-based and hybrid methods. Furthermore, software-based methods are classified into two groups: virtual machine based methods and pure software-based methods. Further, they present some classical software-based systems for multithread deterministic replay debugging. Related to this, we provide a state of the art overview with focus on the processes that may occur during concurrent software debugging.

Hong and Kim present a survey of race bug detection techniques for multithreaded software [60]. They classify 43 race bug and corresponding race bug detection techniques. In addition, they describe and compare the mechanisms of race bug detection techniques. Further, the authors present some examples of race bugs, with the aim to help software developers to avoid race bugs in their code.

Moreover, related to this thesis there are some other studies that propose

taxonomies covering concurrency bug types. Long et al. [61] present a classification of Java concurrency bugs by using a Petri-net model diagram. The transitions in the model represent changes in the concurrent state of a thread. The classification is used to justify the construction of concurrency flow graphs for each method in a concurrent component. The authors believe that the concurrency flow graphs can be used in the construction of test sequences for testing concurrent components to ensure coverage of concurrency primitives.

Tchamgoue et al. [62] classify event-driven program models into low and high level based on event types. They categorize concurrency bug patterns in event-driven programs. In addition to the taxonomy they survey tools for detecting concurrency bugs in these programs. In contrast, our classification of concurrency bugs is based on symptom and system state bug properties.

Helmboldet et al. [63] summarize the concepts of race bug detection techniques for parallel software, and present a taxonomy with respect to the characteristics of the target program structure. Their race taxonomy separates races into categories based on the error types that cause that kind of race (e.g. loop, synchronization operations).

Chapter 4

Research Results

This chapter presents the results of our research in relation to the respective following research goals:

Goal 1: To provide a common terminology for distinguishing between different types and classes of concurrency bugs and to identify the interrelation between separate elements and classes.

Goal 2: To identify the current gaps and less-explored areas in debugging of concurrency bugs.

Goal 3: To identify the current state of concurrency related bugs in real-world software in terms of frequency, severity and resolving time.

4.1 Research Results Related to Goal 1

In order to achieve the first goal of this thesis (*To provide a common terminology for distinguishing between different types and classes of concurrency bugs and to identify the interrelation between separate elements and classes*) we propose a disjoint classification for concurrency bugs by classifying the bugs in a common structure considering the observable properties in paper A [12]. We make use of two types of properties: *System state properties* and *Symptom properties*. Using these properties, we propose a concurrency bug classification. More details about the properties and classification are presented in the following sections.

4.1.1 Concurrent Software Bug Properties

In order to propose this classification, we first gathered the common system states and symptoms properties of bugs based on a literature review. We divide the observable properties in properties related to the system state, and properties related to the symptoms of the concurrent program under test. In the following lists, when we refer to threads t, we are referring to the threads in the set $T_b \subseteq T$, where among all threads T, T_b is the set of threads directly involved in the bug. Similarly, when we refer to a shared resource r, we are referring to a resource in the set $R_b \subseteq R$, where among all resources R, R_b is the set of resources directly involved in the bug.

System State Properties

The below list collects the properties related to the system state at the time of the bug. We refer to the thread execution states (shown in Figure 6.2) in the properties list to present the state of threads when the respective bug occurs. Most of these properties are related to operations of the operating system and they can be observable via available data structures in the operating system kernel, such as Thread Control Block (TCB), or using suitable method(s) in source code to observe these properties during debugging or tracing of the software.

1. At least one thread $t \in T_b$ is in the Waiting state.
2. At least one thread $t \in T_b$ is in the Executing state.
3. At least one thread $t \in T_b$ is in the Ready state.
4. All threads in T_b have read and written to a spinlock variable [1].
5. All threads in T_b are waiting for a lock held by another involved thread.
6. At least one thread $t \in T_b$ is in the ready queue for an unacceptably long time.
7. At least one thread $t \in T_b$ is in Waiting state for an unacceptably long time.
8. All threads in T_b are in Executing state.

[1] spinlock is "mutual execution mechanism in which a process executes in an infinite loop waiting for the value of lock variable to indicate availability" [64]

Symptom Properties

The below list collects the properties related to the observable output at the time of the bug. Based on the bug's symptoms one may recognize the cause of the problem and the nature of the bugs. The following list thus shows some of the typical symptoms that can be used to categorize bugs.

1. No thread $t \in T_b$ is able to proceed and progress.
2. The number of threads in T_b is larger than the number of free processor cores.
3. There are incorrect or unexpected results (e.g., unexpected outputs).
4. The number of requests to a resource r is larger than the number of available resources of that type.
5. All threads in T_b hold a lock.
6. At least one of the threads $t \in T_b$ holds a lock.
7. Accesses to shared memory were made from different threads in T_b.
8. At least one of the accesses to the shared memory was a Write.
9. Accesses to shared memory targeted the same memory location.
10. Accesses to shared memory were NOT protected by a synchronization mechanism.
11. Accesses to shared memory targeted just one memory location.
12. Accesses to shared memory targeted more than one memory location.
13. There were at least two accesses to the same shared memory location, a Write and a Read, where the Read occurred too early.
14. There were at least two Write accesses to shared memory, and they occurred without any Read in-between.
15. There is at least one correct execution ordering between the accesses to shared memory which the program failed to enforce.
16. An atomic execution of statements was required.

Combination of System State and Symptom Properties

Based on the above lists of observable properties, we have derived a classification of concurrency bugs. The resulting classification is shown in Table 4.1. As shown in the table, the first column illustrates the observable properties while the first row displays the different types of concurrency bugs. The mapping between bugs and observable properties should be interpreted as $Bug \rightarrow property$. Thus, an "✓" in the column of bug B and the row of property p would mean that if you have come across bug B, then property p will invariably hold. Note that the reverse implication (i.e., $property \rightarrow Bug$) does not necessarily hold.

Table 4.1: Concurrent software bugs classes and the properties for each class to achieve *Goal 1* (from paper A)

Property	Deadlock	Livelock	Starvation	Suspension	Data race		Order violation			Atomicity violation			
										Single variable		Multi variable	
					Memory inconsistency	Write-Write race	Order violation 1	Order violation 2	Order violation 3	Single variable-AV 1	Single variable-AV 2	Multi variable-AV 1	Multi variable-AV 2
At least one thread $t \in T_b$ is in the Waiting state	✓			✓				✓		✓		✓	
At least one thread $t \in T_b$ is the Executing state		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
At least one thread $t \in T_b$ is in the Ready state			✓						✓		✓		✓
All threads in T_b have read and written to a spinlock variable		✓											
All threads in T_b are waiting for a lock held by another involved thread	✓												
At least one thread $t \in T_b$ is in the ready queue for an unacceptably long time			✓										
At least one thread $t \in T_b$ is in Waiting state for an unacceptably long time	✓			✓									
All threads in T_b are in Executing state					✓	✓	✓						
No thread $t \in T_b$ is able to proceed and progress	✓	✓											
There are incorrect or unexpected results					✓	✓	✓	✓	✓	✓	✓	✓	✓
The number of threads in T_b is larger than the number of free processor cores			✓						✓		✓		✓
Potential request to a resource is larger than the number of available resources of that type				✓									
All threads in T_b hold a lock	✓												
At least one of the threads $t \in T_b$ holds a lock	✓			✓				✓	✓	✓	✓	✓	✓
Accesses to shared memory were made from different threads in T_b					✓	✓	✓	✓	✓	✓	✓	✓	✓
At least one of the memory accesses was Write					✓	✓	✓	✓	✓	✓	✓	✓	✓
Accesses to shared memory targeted the same memory location					✓	✓	✓	✓	✓	✓	✓	✓	✓
The memory accesses were NOT protected by a synchronization mechanism					✓	✓							
Accesses to shared memory targeted just one memory location										✓	✓		
Accesses to shared memory targeted more than one memory location												✓	✓
There were at least two accesses to the same shared memory location, a Write and a Read, where the Read occured too early					✓								
There were at least two Write accesses to shared memory, and they occurred without any Read in-between						✓							
There is at least one correct execution ordering between the memory accesses which the program failed to enforce							✓	✓	✓				
An atomic execution of statements was required										✓	✓	✓	✓

4.1.2 Concurrent Software Bugs

In order to avoid omission of relevant bugs, we conducted a literature review to identify faults, errors and bugs relevant to parallel, concurrent and multicore software testing and debugging. The common properties of bugs presented below are primarily extracted from relevant references based on the literature review.

The explanation of each concurrent bug with their observable properties are listed as follows:

- A **Data race** occurs when at least two threads access the same data and at least one of them write the data [23]. It occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [20] [24].
  - **Memory inconsistency** is when different threads have inconsistent views of shared variables [22]. In this case the results of a write operation by one thread are not guaranteed to be visible to a read operation by another thread.
  - **Write-Write race** is a data corruption caused by accessing a shared variable via at least two threads, in which one of them overwrites the data before any reads.
- **Deadlock** is ”a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs” [65]. More generally, it occurs when two or more threads attempts to access shared resources held by other threads, and none of the threads can give them up [20] [16].
- **Livelock** is ”a situation where a thread is waiting for a resource that will never become available. It is similar to deadlock except that the state of the process involved in the livelock constantly changes with regards to each other, non progressing” [66].
- **Starvation** is ”a condition in which a process indefinitely delayed because other processes are always given preference” [64]. Starvation typically occurs when high priority threads are monopolising the CPU resources.
- A **Suspension-based locking or Blocking suspension** occurs when a calling thread waits for an unacceptably long time in a queue to acquire a lock for accessing a shared resource [67].

- **Order violation** is defined as the violation of the desired order between at least two memory accesses [68]. It occurs when the expected order of interleavings does not appear [44]. If a program fails to enforce the programmer's intended order of execution then an order violation bug could happen [35].

- **Atomicity violation** refers to the situation when the execution of two code blocks (sequences of statements) in one thread is concurrently overlapping with the execution of one or more code blocks of other threads in such a way that the result is inconsistent with any execution where the blocks of the first thread are executed without being overlapping with any other code block. Atomicity violation can be further subcategorized into *single variable atomicity violation* and *multi-variable atomicity violation*, where:

  – **Single variable atomicity violation** is when there is a sequence of concurrent memory access to a single variable, which yields different result from the state of sequential memory accesses [69].

  – **Multi-variable atomicity violation** occurs when multiple variables are involved in an unserializable interleaving pattern [69].

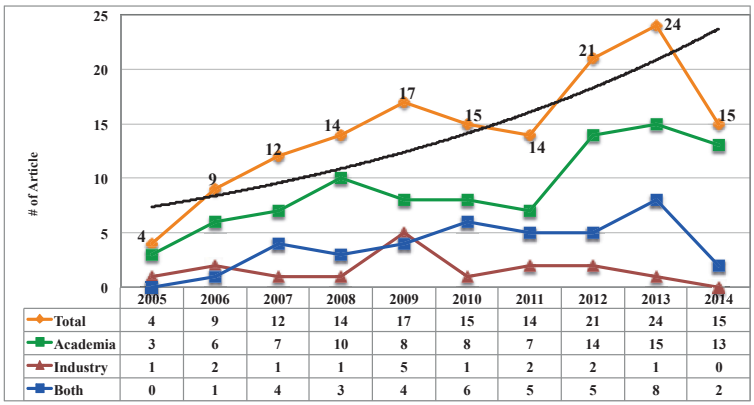
4.2 Research Results Related to Goal 2

In order to achieve the second goal of this thesis (*To identify the current gaps and less-explored areas in debugging of concurrency bugs*) we present the results of a systematic mapping study in the field of concurrent and multicore software debugging in the last decade (2005–2014) in paper B [13].

In terms of publication trends on debugging of concurrent and multicore software during the last decade, we found that the topic has increasingly gained interest since 2005, with the highest number of published papers in 2013. Our investigation indicates that the number of publications in the field increase from 4 in 2005 to 24 in 2013. Figure 4.1 presents the results of our investigation.

In order to investigate the current gaps in debugging concurrency bugs we explored the addressed concurrency bugs, different type of debugging processes, types of research and research contributions.

In term of concurrency bugs we found that six specific types of concurrency bugs (viz., Deadlock, Livelock, Starvation, Data race, Order violation, and Atomicity violation) were addressed by articles in last decade. Among these, a

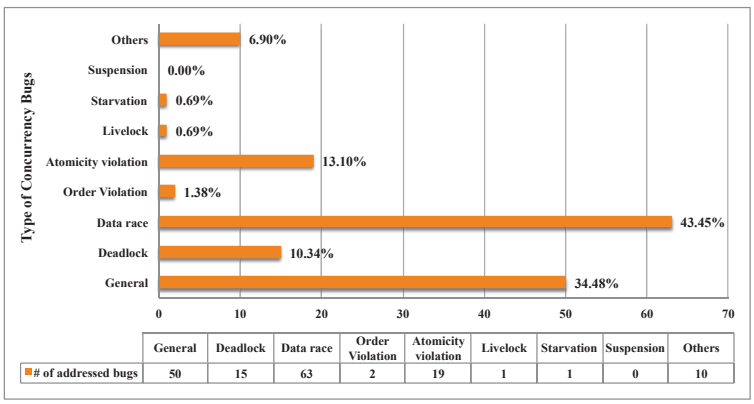


	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014
Total	4	9	12	14	17	15	14	21	24	15
Academia	3	6	7	10	8	8	7	14	15	13
Industry	1	2	1	1	5	1	2	2	1	0
Both	0	1	4	3	4	6	5	5	8	2

Figure 4.1: Distribution of primary study by publication year (from paper B).

large fraction of publications addressed *data race*. More details are presented in Figure 4.2.

Considering different type of debugging procgoal 1esses, we found that five types of debugging process were considered in articles in the period (viz., bug identification, type of bug identification, cause identification, exploring corrections, and fixing bug). Among these, the *bug identification* process as the most common one considered. Figure 4.3 shows the frequency of contributions focusing on different type of the debugging process.



	General	Deadlock	Data race	Order Violation	Atomicity violation	Livelock	Starvation	Suspension	Others
# of addressed bugs	50	15	63	2	19	1	1	0	10

Figure 4.2: Concurrency bugs distribution (from paper B).

Besides, we found that six types of research (viz., validation research, evaluation research, solution proposal, conceptual proposal, opinion paper, and experience paper) in the selected publications, with *solution proposals* being the most common type. The obtained results are illustrated in Figure 4.4.

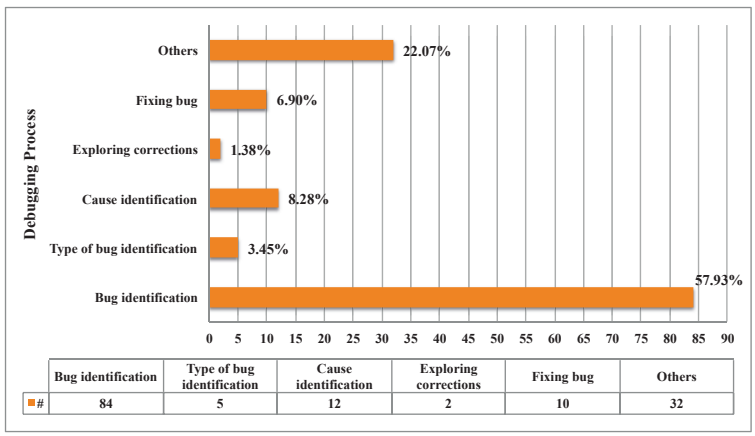


Figure 4.3: Debugging process distribution (from paper B).

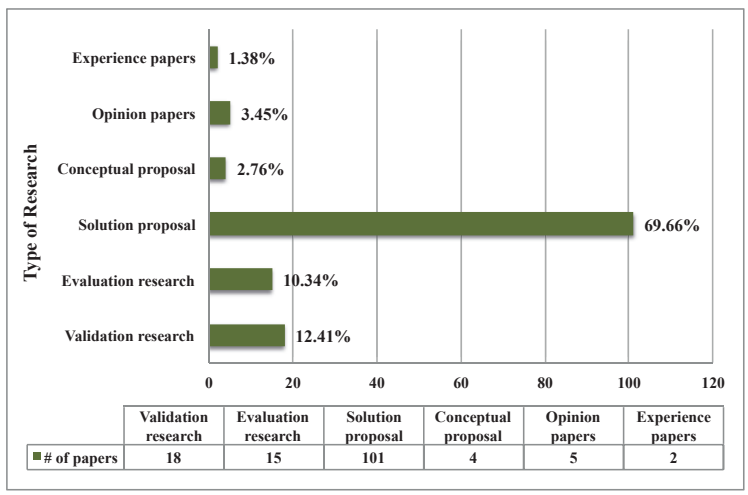


Figure 4.4: Distribution of types of research.

Finally, Figure 4.5 illustrates that the published papers essentially focus on four types of contributions (viz., methods, models, metrics and tools) with *methods* being the most common type.

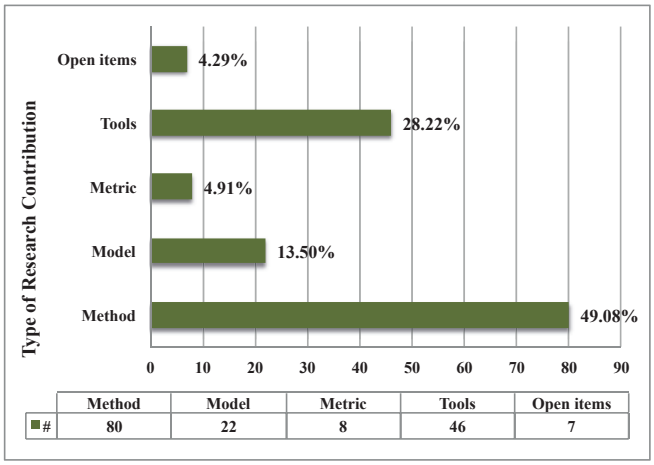


Figure 4.5: Research contribution distribution.

4.3 Research Results Related to Goal 3

In order to achieve the third goal of this thesis (*To identify the current state of concurrency related bugs in real-world software in terms of frequency, severity and resolving time*) we provide a comprehensive study of 4872 fixed bug reports from a widely used open source storage designed for big-data applications (Hadoop [2]). Reported in paper C [14]. The study covers the fixed bug reports from the last decade (2006-2015).

Our comparative study of concurrency bugs and non-concurrency bugs revealed that only 6% of the total set of bugs are related to concurrency issues, while the majority of bugs (i.e., 94%) are of non-concurrency type. The distribution of non-concurrency and concurrency bug types is shown in Figure 4.6.

We also compared the time required to fix concurrency bugs and non-concurrency bugs. Our results show that concurrency bugs do require longer fixing time than non-concurrency bugs, but the difference is not very large.

[2]https://issues.apache.org/jira/browse/hadoop

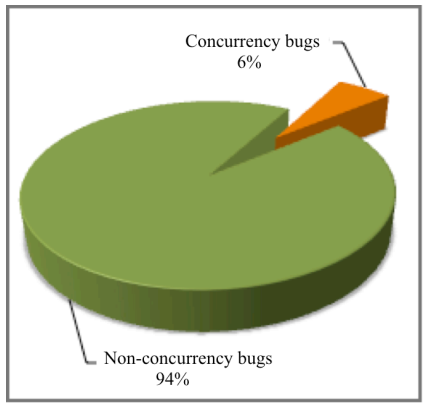


Figure 4.6: Distribution of non-concurrency and concurrency bug types (from paper C).

Figure 4.7 shows the results of comparing the fixing time for concurrency and non-concurrency bugs in the form of box-plots. Boxes span from 1st to 3rd quartile, black middle lines are marking the median and the whiskers extend up to 1.5x the inter-quartile range while the circles represent the outliers.

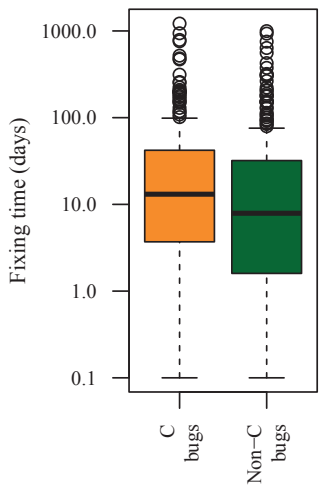
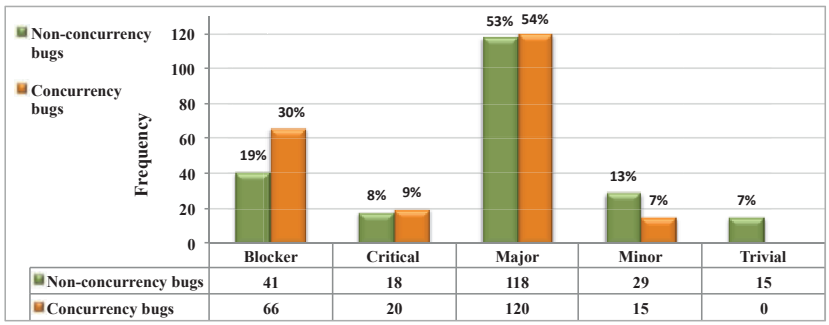


Figure 4.7: Fixing time comparison for concurrency (C) and non-concurrency (Non-C) bugs (from paper C).

Further, our study on severity of concurrency bugs and non-concurrency bugs indicates that concurrency bugs are considered to be more severe than non-concurrency bugs, but the difference is not that large. Figure 4.8 shows the severity distributions.



	Blocker	Critical	Major	Minor	Trivial
Non-concurrency bugs	41	18	118	29	15
Concurrency bugs	66	20	120	15	0

Figure 4.8: Concurrency and non-concurrency bug severity (from paper C).

Chapter 5

Discussion, Conclusion and Future Work

In this chapter, we present a discussion based on our results, a list of conclusions, as well as a set of potential directions for future work.

5.1 Discussion and Limitation

Based on our literature review, we found that existing taxonomies for concurrent and multicore software debugging properties are lacking coverage of some aspects, specifically the ones related to the debugging process. The existing knowledge gaps in different types of bugs may be due to the fact that some specific types of bugs are not well-known yet, or recognizing them is not an easy task.

The results of our systematic mapping study also indicate that researchers from industry have paid less attention to the final steps of the debugging process, i.e., exploring correction and fixing the bugs. It is possible that the initial steps of the debugging process (i.e., bug identification) is considered as the caveat of the process, and once understood and identified, the solutions might be both difficult and involve a lot of work, thus the final steps of the debugging process are not considered novel enough to be worthy of a research publication. Another possibility might be that industry simply cannot involve too detailed aspects of the software and architecture involved in the solution, for public scrutiny thus most refrains from publicly announcing particular solutions.

Some other reasons of these gaps could be that the processes might not be well defined, not applicable in all software development projects, or the process is not easy to apply.

On the other hand, according to our case study results, the distribution of concurrency bugs presented in the bug repository (Hadoop bug report database) is not big. This is not very surprising, since it has long been believed that concurrency bugs are hard to observe and reproduce. There are three main possible reasons for this belief: (1) when users are faced with the bug a single time they may not even be sure that it is a problem with the software and might not report it; (2) it might not be possible to reproduce the bug in the developer's environment due to small differences in the environments even when users are able to reproduce bugs on their machines; (3) software developers might not be able to systematically reproduce the bug using traditional debugging methods since some debugging tools and methods might affect the reproducibility of the bug.

In our case study, we found a much smaller share of concurrency bugs than the one found by other similar studies. This could possibly be due to the one of the three mentioned reason or due to different time span of our study and that of other similar studies. Based on our investigation, 70% of the bugs that we observed were reported in the five-year interval of 2006-2010, and the remaining 30% were reported in the five-year interval of 2011-2015.

Similarly, the fixing time found by other studies is much larger for concurrency bugs than for non-concurrency bugs. We find a difference, but it is relatively small. In our case study we found surprisingly few reports stating difficulties in reproducing the bug. While other studies (e.g., [35]) found that a large portion of fixing time relates to reproducing the bugs and this deference could effect on fixing time calculation.

Moreover, our investigation in the case study shows that about half of the concurrency bugs are of *Data race* type. The reason could be that *Data race* is more severe than other type of bugs and it is quite understandable that it takes longer time to fix.

In the design and execution of this thesis, there are several considerations that need to be taken into account as they can potentially limit the validity of the obtained results. We limited the search for studies and bugs in the systematic study and the case study within the time span of 2005–2014 and 2006–2015, respectively. This was done for two reasons: (1) to limit the volume of search results for practical reasons; (2) to present more *recent* trends (i.e., in the last decade). This limitation of years obviously excludes papers published before the year 2005 and excludes bug reported before the year 2006, including highly

cited papers and important bugs. Thus our systematic mapping study and our case study are not complete with respect to all research papers and reported bugs on the topic, but instead presents the more recent development in the field.

Another threat is related to the classification schema for mapping included papers in our systematic mapping study and included bug reports in our case study. Since authors and bug reporters cannot be expected to follow any standard concurrency bug terminology, partially based on the classification from our grounded theory study we categorized the papers and bug reports. We believe that the process of classification would have been more reliable if consistent terminologies would have been used in the primary studies and bug reports. However, some papers and bug reports were difficult to categorize due to unclear boundaries between some classification scheme categories.

It is possible that the search string and search query may have failed to identify some relevant paper or actual concurrency bugs. It should however be noted that we used more keywords and applied other methods (i.e., backward snowballing and additional secondary search in the systematic research study) compared to previous similar studies.

5.2 Conclusions

We present a grounded theory study. Our study on different types of concurrency bugs proposed a classification of concurrency bugs by classifying the bugs considering their observable properties.

In addition, we provide an overview of existing research on concurrent and multicore software debugging. We also pinpoint current gaps in the research area that may represent opportunities for further research on debugging concurrent and multicore software.

In particular, we provide a case study on concurrency bugs. This study analyzed bugs reported in the Haddop project and provided some evidence of the existence of two classes of bugs: non-concurrency and concurrency bugs. The case study also helped us to recognize the most common types of concurrency bugs in terms of severity and fixing time.

In general, despite all the mentioned limitations, this thesis improves our understanding of the different types of concurrency bugs, the current gaps (or less-explored areas) in debugging concurrency bugs and the current state of concurrency related bugs in real-world software.

5.3 Future Work

This thesis raises a number of questions, which we strongly believe can form the basis of future work, as outlined below.

An interesting agenda for future work would be to combine the evidence identified in the systematic mapping study with evidence from the case study to define hypotheses and theories which will form the basis for proposing new methods, process and tools for concurrent and multicore software debugging. We think a possible future research is to propose solutions to bridge the identified gaps between the paradigms.

Our classification is focusing on shared memory concurrency. There are additional types of concurrency bugs that are specific for message passing systems (e.g., messages race). As a future study we could categorize the primary studies (from our systematic mapping study) and reported bugs (from our case study) based on additional types of concurrency bugs (e.g., those related to message passing).

Moreover, the case study in Chapter 4 provides basis for many research directions. One noticeable research direction is to apply other case studies with other projects (e.g., implemented in other programming languages) in order to generalize the results to other projects.

Another topic for future work, based on this study, could be to conduct a systematic literature review of the field to analyze the existing evidence for concurrent and multicore testing. Also, an interesting additional classification could be related to the domain of different concurrent software testing techniques.

There are still quite a number of issues and aspects that have not been sufficiently covered in the field. It is clear that ensuring the reliability of software, particularly with regard to concurrent and multicore software, will remain a hard problem. Therefore, we believe that software developers and testers will greatly benefit from additional improvements within this field of research.

Bibliography

[1] Hadi Esmaeilzadeh, Emily Blem, Rene St Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Communications of the ACM*, 56(2):93–102, 2013.

[2] David A. Weiser. *Hybrid Analysis of Multi-threaded Java Programs*. ProQuest, 2007.

[3] Jayant Desouza, Bob Kuhn, Bronis R. De Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 78–82. ACM, 2005.

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

[5] Michael Süß and Claudia Leopold. Common mistakes in OpenMP and how to avoid them. In *OpenMP Shared Memory Parallel Programming*, pages 312–323. Springer, 2008.

[6] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 245–254. ACM, 2010.

[7] Joab Jackson. Nasdaq's Facebook Glitch Came From Race Conditions, May 2012. preprint (2011), available at `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[8] Johanna Schneider. Tracking down root causes of defects in simulink models. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, page 1. ACM, 2014.

[9] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[10] Colin Robson. *Real world research: A resource for social scientists and practitioner-researchers*, volume 2. Blackwell Oxford, 2002.

[11] D.E. Perry, S.E. Sim, and S.M. Easterbrook. Case studies for software engineers. In *26th International Conference on Software Engineering, ICSE 2004*, pages 736–738, May 2004.

[12] Sara Abbaspour A, Hans Hansson, Daniel Sundmark, and Sigrid Eldh. Towards Classification of Concurrency Bugs Based on Observable Properties. In *Workshop on Complex faUlts and Failures in LargE Software Systems (COUFLESS)*, 2015.

[13] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Wasif Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Software Quality Journal*, pages 1–34, 2016.

[14] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Eduard Paul Enoiu. A study of concurrency bugs in an open source software. In *Proceedings of the 12th International Conference on Open Source Systems (OSS)*, 2016.

[15] R.W. Brown. Method and apparatus for processing requests for video presentations of interactive applications in which vod functionality is provided during nvod presentations, June 23 1998. US Patent 5,771,435.

[16] Darryl Gove. *Multicore Application Programming: For Windows, Linux, and Oracle Solaris*. Addison-Wesley Professional, 2010.

[17] Juan Gonzalez, David Insa, and Josep Silva. A new hybrid debugging architecture for eclipse. In *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation(LOPSTR)*, pages 183–201. Springer International Publishing, 2014.

[18] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 284–295. IEEE Computer Society, 2005.

[19] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[20] K. Henningsson and C. Wohlin. Assuring fault classification agreement - an empirical evaluation. In *2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04. Proceedings*, pages 95–104, August 2004.

[21] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145. ACM, 2008.

[22] Leon Li Wu and Gail E. Kaiser. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. Technical report, Columbia University, 2011.

[23] Noriaki Yoshiura and Wei Wei. Static data race detection for java programs with dynamic class loading. In *Internet and Distributed Computing Systems*, pages 161–173. Springer, 2014.

[24] Shameen Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006.

[25] 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 *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 611–622. ACM, 2014.

[26] Ghazia Zaineb and Irfan Anjum Manarvi. Identification And Analysis Of Causes For Software Bug Rejection With Their Impact Over Testing Efficiency. *International Journal of Software Engineering & Applications (IJSEA)*, 2(4), 2011.

[27] Ryan Eccles, Blair Nonneck, Deborah Stacey, and others. Exploring parallel programming knowledge in the novice. In *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pages 97–102. IEEE, 2005.

[28] Ryan Eccles, Deborah Stacey, and others. Understanding the parallel programmer. In *High-Performance Computing in an Advanced Collaborative Environment, 2006. HPCS 2006. 20th International Symposium on*, page 12. IEEE, 2006.

[29] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Shadnaz Asgari, Victor Basili, Jeffrey K. Hollingsworth, and Marvin V. Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 conference*, pages 35–35. IEEE, 2005.

[30] Jan Lönnberg and Anders Berglund. Students' understandings of concurrent programming. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*, pages 77–86. Australian Computer Society, Inc., 2007.

[31] Jan Lönnberg, Lauri Malmi, and Mordechai Ben-Ari. Evaluating a visualisation of the execution of a concurrent program. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 39–48. ACM, 2011.

[32] Sebastian Nanz, Scott West, and Kaue Soares Da Silveira. Examining the expert gap in parallel programming. In *Euro-Par 2013 Parallel Processing*, pages 434–445. Springer, 2013.

[33] Jan Lönnberg, Lauri Malmi, and Anders Berglund. Helping Students Debug Concurrent Programs. In *Proceedings of the 8th International Conference on Computing Education Research*, Koli '08, pages 76–79, New York, NY, USA, 2008. ACM.

[34] Caitlin Sadowski and Jaeheon Yi. User Evaluation of Correctness Conditions: A Case Study of Cooperability. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 2:1–2:6, New York, NY, USA, 2010. ACM.

[35] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[36] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 426–438. ACM, 2015.

[37] J. Schimmel, K. Molitorisz, and W.F. Tichy. An evaluation of data race detectors using bug repositories. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 1361–1364, Sept 2013.

[38] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*, volume 1 of *ICSE '10*, pages 485–494, May 2010.

[39] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. *Microsoft Research*, 38:39, 2007.

[40] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[41] Gowritharan Maheswara, Jeremy S. Bradbury, and Christopher Collins. Tie: An interactive visualization of thread interleavings. In *Proceedings of the 5th international symposium on Software visualization*, pages 215–216. ACM, 2010.

[42] Steven P. Reiss and Manos Renieris. Demonstration of JIVE and JOVE: Java as it happens. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 662–663. IEEE, 2005.

[43] Steven P. Reiss and Suman Karumuri. Visualizing threads, transactions and tasks. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–16. ACM, 2010.

[44] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

[45] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. UNICORN: a unified approach for localizing non-deadlock concurrency bugs. *Software Testing, Verification and Reliability*, 25(3):167–190, 2015.

[46] Sangmin Park, Mary Jean Harrold, and Richard Vuduc. Griffin: grouping suspicious memory-access patterns to improve understanding of concurrency bugs. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 134–144. ACM, 2013.

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

[48] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[49] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

[50] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, volume 10, pages 1–16, 2010.

[51] Armin Biere Cyrille Artho, Klaus Havelund. High-level data races. In *journal on software testing, verification and reliability (STVR)*, pages 1–12, 2003.

[52] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 1–14, 2005.

[53] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48. ACM, 2006.

[54] Ruirui Huang, Erik Halberg, and G. Edward Suh. Non-race concurrency bug detection through order-sensitive critical sections. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 655–666, New York, NY, USA, 2013. ACM.

[55] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336. ACM, 2014.

[56] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262. ACM, 2012.

[57] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages; Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[58] Maria Brito, Katia R. Felizardo, Paulo Souza, and Simone Souza. Concurrent Software Testing: A Systematic Review. *on Testing Software and Systems: Short Papers*, page 79, 2010.

[59] Peng Wang, Xiaofang Qi, Xiaoyu Zhou, and Xiang Zhang. Multithread Deterministic Replay Debugging: The State of The Art. *International Journal of Advancements in Computing Technology*, 4(23), 2012.

[60] Shin Hong and Moonzoo Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.

[61] B. Long and P. Strooper. A classification of concurrency failures in java components. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, April 2003.

[62] G.M. Tchamgoue, O.-K. Ha, K.-H. Kim, and Y.-K. Jun. A taxonomy of concurrency bugs in event-driven programs. In *Communications in Computer and Information Science*, volume 257 CCIS, pages 437–450, 2011.

[63] D.P. Helmbold and C.E. McDowell. A taxonomy of race conditions. *J. Parallel Distrib. Comput.*, 33(2):159–164, March 1996.

[64] William Stallings. *Operating Systems- internals and design principles*, volume 7th. Prentice Hall Englewood Cliffs, 2012.

[65] Yogesh Bhatia and Sanjeev Verma. Deadlocks in distributed systems. *International Journal of Research*, 1(9):1249–1252, 2014.

[66] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[67] Shiyao Lin, Andy Wellings, and Alan Burns. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. *Concurrency and Computation: Practice and Experience*, 25(16):2227–2251, 2013.

[68] Deepal Jayasinghe and Pengcheng Xiong. CORE: Visualization tool for fault localization in concurrent programs. 2010.

[69] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 51–60. IEEE, 2012.