Effective Testing for Concurrency Bugs

Thesis for obtaining the title of

Doctor of Engineering Science of the Faculties of Natural Science and Technology I of Saarland University

From

Pedro José Sousa da Fonseca

Saarbrücken, 13. April 2015

Date of Colloquium:	24/06/2015
Dean of Faculty:	Univ.-Prof. Dr. Markus Bläser
Chair of the Committee:	Prof. Dr. Andreas Zeller
First Reviewer:	Prof. Dr. Rodrigo Rodrigues
Second Reviewer:	Prof. Dr. Peter Druschel
Third Reviewer:	Prof. Dr. George Candea
Forth Reviewer:	Dr. Björn Brandenburg
Academic Assistant:	Dr. Przemyslaw Grabowicz

Dedicated to my parents and brother

This work was supported by the
Foundation for Science and Technology of Portugal (SFRH/BD/45309/2008)
and the
Max Planck Society

Abstract

In the current multi-core era, concurrency bugs are a serious threat to software reliability. As hardware becomes more parallel, concurrent programming will become increasingly pervasive. However, correct concurrent programming is known to be extremely challenging for developers and can easily lead to the introduction of concurrency bugs. This dissertation addresses this challenge by proposing novel techniques to help developers expose and detect concurrency bugs.

We conducted a bug study to better understand the external and internal effects of real-world concurrency bugs. Our study revealed that a significant fraction of concurrency bugs qualify as semantic or latent bugs, which are two particularly challenging classes of concurrency bugs. Based on the insights from the study, we propose a concurrency bug detector, PIKE that analyzes the behavior of program executions to infer whether concurrency bugs have been triggered during a concurrent execution. In addition, we present the design of a testing tool, SKI, that allows developers to test operating system kernels for concurrency bugs in a practical manner. SKI bridges the gap between user-mode testing and kernel-mode testing by enabling the systematic exploration of the kernel thread interleaving space. Our evaluation shows that both PIKE and SKI are effective at finding concurrency bugs.

Kurzdarstellung

Im gegenwärtigen Multicore-Zeitalter sind Fehler aufgrund von Nebenläufigkeit eine ernsthafte Bedrohung der Zuverlässigkeit von Software. Mit der wachsenden Parallelisierung von Hardware wird nebenläufiges Programmieren nach und nach allgegenwärtig. Diese Art von Programmieren ist jedoch als äußerst schwierig bekannt und kann leicht zu Programmierfehlern führen. Die vorliegende Dissertation nimmt sich dieser Herausforderung an indem sie neuartige Techniken vorschlägt, die Entwicklern beim Aufdecken von Nebenäufigkeitsfehlern helfen.

Wir führen eine Studie von Fehlern durch, um die externen und internen Effekte von in der Praxis vorkommenden Nebenläufigkeitsfehlern besser zu verstehen. Diese ergibt, dass ein bedeutender Anteil von solchen Fehlern als semantisch bzw. latent zu charakterisieren ist – zwei besonders herausfordernde Klassen von Nebenläufigkeitsfehlern. Basierend auf den Erkenntnissen der Studie entwickeln wir einen Detektor (PIKE), der Programmausführungen daraufhin analysiert, ob Nebenläufigkeitsfehler aufgetreten sind. Weiterhin präsentieren wir das Design eines Testtools (SKI), das es Entwicklern ermğlicht, Betriebssystemkerne praktikabel auf Nebenäufigkeitsfehler zu uberprụfen. SKI füllt die Lücke zwischen Testen im Benutzermodus und Testen im Kernelmodus, indem es die systematische Erkundung der Kernel-Thread-Verschachtelungen erlaubt. Unsere Auswertung zeigt, dass sowohl PIKE als auch SKI effektiv Nebenläufigkeitsfehler finden.

Acknowledgements

I'm greatly indebted to my advisor, Rodrigo Rodrigues, for his continuous effort over the years to support and encourage me. His exceptional energy and constructive optimism were absolutely essential to the success of this work. His altruism made me a better person.

This thesis would not exist without the contributions of my collaborators: Björn Brandenburg, Cheng Li and Vishal Singhal. Special thanks to Björn for the feedback and support provided during the last few years of my work.

I won't forget that, during my first years at MPI-SWS, I learned a lot with the perseverance and experience of Atul Singh and Andreas Haeberlen. The energy of Pramod Bhatotia and Cheng Li was contagious. Thank you all.

My friends and colleagues at MPI-SWS, in particular the SysNets group members, had a crucial role in producing an inspiring research environment and providing invaluable feedback that were critical to this research. Living in Saarbrüecken, while working on my dissertation, was an enjoyable experience, in large part, thanks to my friends.

I am grateful to my parents and my brother for their permanent support throughout my life.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Hardware parallelism: A blessing and a curse

There is an ongoing and fundamental change in hardware. In the past few decades, the performance of processors has been consistently increasing at exponential rates [EBA+13], mostly dominated by substantial increases in the density of transistors and in the operating frequency of processors [BC11]. However, since the mid-2000s hardware architects have been facing serious challenges that have threatened the processor performance growth that consumers and system designers have come to expect and rely upon.

Recently, the high frequency of processors has reached a point that energy consumption and heat dissipation have become major obstacles hindering frequency scaling [Ant14]. Because of these obstacles, hardware architects have been unable to continue ramping the frequency of processors like they had been doing for decades. Instead, hardware architects have now been forced to rely on other strategies to sustain performance growth.

To address these challenges, hardware architects have recently been increasing hardware parallelism by shifting from a frequency scaling strategy to a multicore scaling strategy. Consequentially, currently most commodity processors already consist of several cores and major processor manufacturers have plans to further increase the number of cores in subsequent generations of processors [ABD+09]. But shifting to multicore scaling is a significant departure from previous growing strategies because, in contrast with other design strategies (e.g. frequency scaling), hardware parallelism is not transparent to software.

The lack of transparency of hardware parallelism, as expected, is having a profound impact on software and on programmers: the multi-core era represents an unprecedented shift in paradigm with regard to the way software is built, maintained and executed. To take advantage of parallel hardware, software is expected to become increasingly parallel itself [HM08]. However, developing parallel software is well known to be a very challenging task for programmers and prone to concurrency bugs [SBN+97].

1.2 The threat of concurrency bugs

It is challenging to develop parallel software. Developing parallel software requires the programmer to keep track of all the possible communication patterns due to the, typically, extremely large number of possible interleavings between different tasks. While it is possible to create parallel applications using different communication paradigms, in this work we focus on the multi-threading paradigm, which relies on the use of threads and on the shared memory model, that has been found to be a practical and popular programming model. For multi-threaded applications, the communication between threads and the large set of possible interleavings significantly increase the level of complexity of software development. When programmers fail to program correctly a multi-threaded application, they run the risk of introducing concurrency bugs in software, which are a particularly challenging class of bugs.

Concurrency bugs constitute an important class of bugs. Concurrency bugs are only triggered when multi-threaded software simultaneously receives certain inputs and is executed under certain interleavings of instructions, as chosen by the operating system scheduler. It is the dependency on the interleaving of instructions that sets apart concurrency bugs from non-concurrency bugs. This key difference is responsible for the non-deterministic nature of concurrency bugs, and, it is the reason why concurrency bugs are also referred to as Heisenbugs [Gra86].

The non-deterministic nature of concurrency bugs makes concurrency bugs hard to find, hard to reproduce and hard to fix. Because these bugs are non-deterministic it is hard for developers to come across concurrency bugs using traditional testing methodologies. Even when developers do witness the application's misbehavior it is hard for them to reproduce the bug by re-executing the application given that it requires the operating system to choose again the same or equivalent thread interleavings, while keeping the original environment. And when developers are able to find and reproduce concurrency bugs it is still challenging to apply a fix and, unfortunately, easy to (re)introduce the same or different concurrency bugs.

In addition to being challenging for developers, concurrency bugs are also known to cause significant damage to users and are considered to be a serious treat to software reliability. Prominent examples of serious harm caused by concurrency bugs include the 2003 large-scale US blackout [GEE], which affected millions of people, and the fatal Therac-25 incident [LT93], which caused loss of human lives. Given that parallel software is expected to have an increasingly prominent role in computer systems, it is imperative to provide programmers with tools and methodologies to help them to better handle concurrency bugs [ABD+09].

1.3 Problem statement

This dissertation addresses two important problems that arise in the context of testing large-scale software for concurrency bugs: detecting bugs and exposing bugs.

Detecting bugs consists in distinguishing correct software behavior from incorrect behavior. Some bugs have obvious manifestations, which simplifies detection during testing, for instance, bugs that lead to the execution of illegal instructions are trivially detected. However, because other bugs have subtle effects, such as providing a syntactically correct response but with incorrect content, bug detection can be exceptionally hard.

Because of the size of current software and its complexity, in general, it is not feasible to ask developers to manually write the entire specification of the software. Therefore detecting concurrency bugs using the explicit specification approach, which would be the most effective solution assuming a correct and complete specification, is in practice in the case of the most complex software.

An alternative to manually specifying the correct behavior of software is to rely on manual inspection of their behavior by the programmers themselves. Unfortunately, some concurrency bugs are not easily detected by the programmers because they lead to non-obvious failures or because it is hard to witness the failures. For example, the application might simply return stale results to users, as opposed to returning error or warning messages. In these difficult cases, without automatic bug detection mechanisms, the programmer does not easily notice that an instance of incorrect behavior occurred. Similarly, even the software users may be unable to promptly detect such concurrency bugs, even when triggered. For this reason, effective bug detection mechanisms are important to test concurrent software.

In this work we developed a new systematic concurrency bug detection approach that targets particularly challenging classes of bugs. The basic idea of our approach is to detect concurrency bugs by executing multi-threaded applications and analyzing their behavior under different thread interleavings. More specifically, our approach compares a given execution, when certain requests are executed concurrently, with the corresponding linearized executions [HW90] of the same requests. To understand the viability of this approach and to guide the development of our bug detector, we conducted a study of real-world concurrency bugs.

Our study resulted in the discovery of several cases of concurrency bugs which manifest themselves in non-obvious ways – i.e., by providing wrong results to the user (semantic concurrency bugs). This type of misbehavior contrasts with the one of other concurrency bugs which instead produce immediate effects that are easily detected by the developer and even the users. A typical example of an easily detectable misbehavior is an application crash. In addition, the study also lead to the discovery of several cases of concurrency bugs that are not immediately exposed to the user when triggered (latent concurrency bugs). This latter class of concurrency bugs is also hard to detect because it is difficult to expose the effects of these bugs to developers, since even after being triggered their effects may not be externally visible.

Because these two classes of concurrency bugs have been found to impair real-world applications and because there is a lack of effective approaches to detect them, it is important to develop better detection approaches. This dissertation addresses this problem by presenting a novel detection technique that targets semantic and latent concurrency bugs, having the advantage of not requiring the programmer to specify the correct behavior of the application. Our approach consists of implicitly extracting a specification of the application by testing if the application obeys linearizable semantics [HW90]. Intuitively, linearizability means that the application, when executing concurrent requests, behaves as if the requests were executed serially, in some order that is consistent with the real-time ordering of the invocations and replies to the requests.

By systematically testing if linearizability is upheld, our approach is able to find subtle violations of the application semantics. Furthermore, by checking if both the output and the internal state of the application obey the inferred semantics, our approach is capable of identifying not only the bugs that manifest themselves immediately as a wrong output (semantic bugs), but also those that silently corrupt internal state (latent bugs).

Apart from the problem of detecting concurrency bugs during an execution, this dissertation addresses a second important problem which is the need to **expose concurrency bugs**, that is, the need to generate executions that cause concurrency bugs to become visible to bug detectors. To generate such executions, and given the schedule dependency of concurrency bugs, concurrency testing techniques have to explore the interleaving space of the application.

In general, exploring the thread interleaving space is challenging because of the vast size of this space. For complex applications it is not feasible to exhaustively perform this exploration. Instead of exhaustively exploring the interleaving space, there are better approaches that perform the interleaving space exploration cleverly, by avoiding redundancies and prioritizing the most important points in this space.

Systematic approaches have been proposed for testing user-mode applications. However, unfortunately, we are not aware of any previous tool that accomplishes systematic exploration of the interleaving space for real-world kernels – despite the fact that kernels are typically highly complex and absolutely critical for the reliability of systems. While in user-mode there exist tools for the purpose of exploring the thread interleaving space, it is challenging to directly apply these tools to the kernel because such approach would require modifications to the kernel. We believe that requiring developers to modify the kernel would become a major obstacle in the adoption of these testing tools, as it would require an initial development effort and also a significant maintenance effort, as the kernel evolves. Instead, this dissertation explores a different design by proposing a practical kernel testing approach that is effective and simultaneously avoids the need for kernel modifications.

This kernel testing approach controls the interleaving of the kernel threads by relying on a modified virtual machine monitor. Using standard OS mechanisms to associate threads to specific virtual processors (i.e., processor affinity), it relies on using a virtual machine monitor to control the relative speed of different threads, by independently suspending and resuming the execution of machine instructions for each of the virtual processors. This is a convenient mechanism to control the progress of threads but it is not by itself sufficient to implement state-of-the-art thread interleaving exploration algorithms [BKMN10, MQB$^+$08]. Existing algorithms were not intended to be applied to kernels and so do not address interrupts which are an important source of concurrency in kernels. Therefore these algorithms need to be extended to be effective for testing kernels. Another challenge arises from the fact that existing algorithms, apart from requiring a method to exercise control over the progress of threads, additionally require information about the liveness of individual threads. To gather this information our approach relies on the use of heuristics that analyze the execution of instructions to infer the state of threads (e.g., whether the CPU is actually making progress or whether the CPU is waiting on a spin-lock).

1.4 Contributions

This dissertation makes the following main contributions to the state-of-the-art:

- A study on the effects of real-world concurrency bugs. The definition, along with results that show the existence, of two complex and orthogonal types of concurrency bugs: semantic concurrency bugs and latent concurrency bugs.

- A set of guidelines to annotate the state of complex applications that enables state comparison between different executions of the application.
- A method to infer the specification of complex applications by analyzing the behavior of the application when executed under different thread interleavings. The inferred specification can then be used to detect concurrency bugs during application executions. In addition, we found concrete evidence that MySQL, an important and complex application, in the general case is expected to provide linearizability semantics at the level of user requests.
- A method, based on virtual machines, to systematically explore the interleaving space of unmodified operating system kernels and a set of techniques to efficiently achieve fine-control of the thread interleavings.

Furthermore, in the context of the work presented in this dissertation, two systems were implemented:

- PIKE: A bug detector that detects both semantic concurrency bugs and latent concurrency bugs.
- SKI: A testing tool that exposes concurrency bugs in operating system kernels using a virtual machine monitor.

1.5 Outline

The rest of this dissertation[1] is structured as follows. Chapter 2 discusses the related work and Chapter 3 discusses important background. Chapter 4 presents an overview of the methodology and the results of the study on the effects of real-world concurrency bugs. Chapter 5 presents our bug detection proposal that targets semantic and latent concurrency bugs. Chapter 6 presents our approach to systematically explore the interleaving space of kernels. Chapter 7 discusses several important aspects of testing tools, in general, and describes our experience interacting with developers and applying our tools to the real-world. Chapter 8 discusses future research directions and finally we conclude the dissertation with Chapter 9.

[1]This dissertation incorporates and extends work previously published [FLSR10, FLR11, FRB14]. Namely, this work additionally includes Chapter 7, Chapter 8 and a general consolidation and revision of the text.

2 Related work

This chapter relates the work presented in this dissertation with previous work. First we discuss existing work on the characteristics of bugs and in the context of detecting and exposing bugs. We then discuss previous work on deterministic replay and virtual machine introspection techniques, which is work specifically related to SKI.

2.1 Studies of bugs

Chapter 4 presents a study of real-world concurrency bugs. A series of related studies have been conducted before and after ours. In particular, there is a large body of literature about the propagation [VT06, PLM+10a] and even prediction [NZHZ07, NB05, RKBD14, LLS+13] of bugs in source code. Some of these studies use the revision control system to understand the behavior of programmers and its effects on software reliability (e.g., which components or source code files are most prone to errors). This work is complementary to the study conducted in this dissertation, which is focused on a specific class of bugs (i.e., concurrency bugs) and on understanding their consequences.

In previous work, researchers analyzed the consequences of bugs for three different database systems [VBLM07]. However the authors did not distinguish between concurrency and non-concurrency bugs, and only evaluated whether they caused crash or Byzantine faults, since that work was focused on presenting a replication architecture, instead of being focused on studying bugs.

Chandra and Chen [CC00] looked at the bug databases of three open-source applications (Apache web server, GNOME desktop environment and MySQL database) but the focus of their work was quite different from ours. They analyzed all bugs (among which only 12 were concurrency bugs) and focused exclusively on determining whether generic recovery techniques such as process pairs would be effective in tolerating them. In their case, concurrency bugs were only one possible type of bug that fell into the category for which such techniques are effective. In contrast, we focus on a more narrow class of bugs by limiting ourselves to concurrency bugs, but provide a broader analysis taking into consideration several characteristics of these bugs.

More recently, in a study focused on file systems [LADADL13], researchers analyzed the development patches, which include but are not limited to bug fixes, of six file systems to gather information about several broad aspects of their development. Among these aspects, the authors analyzed the type of patches (e.g., whether they contribute with documentation, new features or bug fixes). Their analysis on bug fixing patches included semantic bugs and concurrency bugs, but not concurrency semantic bugs nor concurrency latent bugs, which is an important aspect analyzed in our study.

The reliability of older database systems was carefully studied by Sullivan and Chillarege [SC92]. In this study, Sullivan and Chillarege analyzed the DB2 and IMS database systems, as well as the MVS operating system, by inspecting reports filed by dedicated field personnel. The focus of this study was not on concurrency, but concurrency was also analyzed and the authors concluded that the user interface of the database was a typical source of concurrency bugs.

Farchi et al. analyzed concurrency bugs, but by artificially creating them [FNU03]. The methodology adopted by the study was to ask programmers to write programs containing concurrency bugs, which arguably may not lead to bugs that are representative of real-world problems. In contrast, we analyze a database of bugs in a widely used, well-maintained application.

Lu et al. [LPSZ08] studied real concurrency bugs that were found in four open-source applications. Using the respective bug report databases, the authors analyzed a total of 105 concurrency bugs. 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 build on this study, in particular by using a very similar methodology for deciding which bugs to analyze, but provide a complementary angle by studying the effects of concurrency bugs (e.g., whether concurrency bugs are latent or not, or what type of failures they cause).

Sahoo et al. analyzed the reproducibility of bugs [SCA09, SCA10]. While the main focus of their study was not concurrency bugs, the authors distinguished concurrency bugs from non-concurrency bugs when trying to characterize their reproducibility. The study analyzed several server applications focusing on the properties of the inputs that are required to trigger bugs. Interestingly their study concluded that a large percentage of bugs, 77%, can be triggered by just providing one input after the client establishing a session with the server.

Lin et. al [TLL+14] analyzed bugs found in the Linux kernel and in two user-mode applications (the Mozilla browser and the Apache web server). Their study concluded that the kernel has a significantly higher fraction of concurrency bugs (13.6%), in comparison with the user mode applications studied (1.2% for Mozilla and 5.2% for Apache), and suggest that it may be caused by the complexity of the kernel synchronization. Furthermore the authors found that 10.2% of all kernel bugs studied are related to interrupt handling, such as missing instructions to enable or disable interrupts at the appropriate locations.

Other researchers have studied bugs in the context of other concurrency paradigms, namely large-scale data processing frameworks (e.g., MapReduce [LZL+13a] and SCOPE [KTGN10]). In contrast, the focus of this dissertation is on shared memory multi-threaded applications.

2.2 Testing software

Testing software is a very difficult task and many different approaches have been proposed in the past to address this challenge. Testing tools can be broadly divided into static analysis and dynamic analysis tools depending, respectively, on whether they simply analyze the source code or actually execute the code. Furthermore, a few hybrid tools attempt to leverage the advantages of both of these approaches by having both a static and a dynamic analysis component [KZC13, CLL+02, EQT07, SC07].

Static tools such as RacerX [EA03] and others [NAW06, BLR02] have the advantage of not being limited in their analysis to the execution path determined by the input. In contrast, dynamic analysis tools, since they actually run the code, have the advantage of having more information about the context of the execution and therefore can potentially achieve a higher accuracy (i.e., fewer false positives). SKI and PIKE are examples of dynamic analysis tools. Other examples include FastTrack [FF09], LiteRace [MMN09] and Eraser [SBN+97].

Because dynamic testing tools run the tested applications and bugs might only manifest under certain inputs and/or under certain interleavings of threads, dynamic testing approaches require a method to explore different control and data paths [RW85], and a method to analyze those executions, to detect bugs in the executions. Section 2.2.1 discusses in detail several proposed techniques to detect concurrency bugs and Section 2.2.2 discusses previous work in the context of exposing concurrency bugs.

2.2.1 Detecting concurrency bugs

Given the specifics of concurrency bugs, researchers have developed different types of bug detectors to handle this special class of bugs. In this section, we discuss existing concurrency bug detectors, including data race detectors and linearizability checkers, which are two important classes of concurrency bug detectors.

Data race detectors

Data races occur when pairs of data memory accesses, in which at least one access is a write, are not synchronized and can be executed at the same time. Although data races are not necessarily bugs and not all concurrency bugs are data races, experience has shown that data races are dangerous and can easily lead to application behavior that is unintended by the developers, namely concurrency bugs. Thus, many data race detectors have been proposed with the goal of detecting concurrency bugs [SBN+97, VCFN11, LTQZ06, FF04, YRC05, BCM10, MMN09, FLR11, LOCb, EMBO10].

Data race detectors can be roughly divided into two types depending on which algorithm they use. The first type of data race detector relies on the lockset algorithm [SBN+97] to infer whether the programmer protected all accesses to a specific shared variable with a common lock. The second type of data race detector relies on the happens-before algorithm [FF09, MMN09]. Recently, Erickson et al. have proposed a different data race detector, DataCollider, that is not based on either of these algorithms, but is instead based on sampling and the use of breakpoints [EMBO10]. In Section 6.5.5 we explain how SKI combines the approach of DataCollider, to detect kernel data races, with a novel method to expose kernel concurrency bugs.

Like PIKE, data race detectors are tools that are able to be detect concurrency bugs, however they have distinct features. First, data race detectors detect data races instead of directly detecting concurrency bugs. Since programs often contain benign data races, simply detecting data races easily leads to false positives. Furthermore the absence of data races is not a guarantee of correct synchronization [AHB03, LTQZ06], and hence false negatives can result. Another difference is that race detectors typically operate at the lower-level of individual memory accesses. In contrast, PIKE analyzes the actual output of the application as well as a high-level digest of the state, potentially uncovering bugs that are not triggered by low-level data races and also facilitating the process of inspecting the results.

In order to reduce the number of false positives in data race finding tools and thus reduce the burden on testers, researchers have developed heuristics. By using heuristics some systems attempt to identify scenarios that frequently lead to false positives. DataCollider [EMBO10], for example, tries to detect benign data races caused by counters and accesses to different bits of the same variable. One approach is to use heuristics that rely on looking at the instructions at or near the problematic accesses or on manually white-listing variables. The disadvantage of this approach is that it also increases the risk of missing erroneous data races. Another interesting approach to distinguish erroneous data races from benign data races relies on replaying the execution [NWT+07]. It attempts to trigger the opposite outcome of the data race and then comparing the low-level results obtained with both data race outcomes. This approach, however, still aims at finding low-level data races. RedFlag [UBH09] is another example of a concurrency bug detector for the kernel that combines a block-based atomicity checker [WS06] with a lockset-based data race detector [SBN+97].

Linearizability checkers

PIKE uses linearizability as a specification and differs from previous proposals [BDMT10, XBH05a, VYY09, Vaf10] that use this approach in two ways. First, previous approaches typically ignore the internal state of the application, which is important for the detection of latent bugs. Second, previous approaches check for the atomicity of smaller sections of code such as code blocks or library calls, which poses fewer challenges than testing the linearizability of large server applications and does not allow for the detection of some concurrency bugs.

AVIO [LTQZ06] detects atomicity violations at the level of individual memory accesses. AVIO achieves this by learning from a large set of runs (which are assumed to be correct) the valid memory access patterns (e.g., when are two consecutive accesses from a thread allowed to be interleaved by an access from another thread). AVIO shares our goal of attempting to find concurrency bugs without relying on finding data races, but in contrast AVIO works at a low-level and relies on training.

Other detectors

Huang et. al proposed a bug detector [HHS13] that identifies pairs of critical sections that non-deterministically change the contents of shared memory, depending on their execution order. This analysis, in essence, relies on detecting whether critical sections are commutative.

CAFA [HYN+14] finds concurrency bugs in the context of event-driven programming frameworks for mobile devices. CAFA leverages a causality model of the underlying mobile operating system to find concurrency bugs caused by *use-after-free* violations. The causality model enables the tool to understand the dependencies between events and, therefore, limits the false positives. Other researchers have addressed the problem of detecting concurrency bugs in different types of event-based frameworks [PVSD12, RVS13].

2.2.2 Exposing concurrency bugs

Dynamic testing requires executing non-concurrent applications with a diverse set of inputs to ensure high testing coverage and, therefore, increase the chances of exposing bugs. In addition to testing with a diverse set of inputs, concurrent applications also need to be tested with a diverse set of thread interleavings given that concurrency bugs are only exposed under a subset of all possible interleavings. This section describes the related work in the context of exposing concurrency bugs, with regard to both interleaving and input space exploration.

Interleaving space exploration

When a multi-threaded application runs natively, the operating system tends to choose similar thread interleavings for different executions. Because the interleaving diversity is important to ensure the effectiveness of testing concurrent applications, many approaches have been proposed to address this problem.

The traditional approach for increasing the diversity of interleavings is to rely on stress testing – i.e., repeatedly run the tested application several times hoping that eventually a large and representative set of interleavings is explored. To further increase the effectiveness of this technique, several methods have been proposed to augment stress testing with noise generators [BAEFU06b, BAEFU06a, Sto02, PLZ09, EMBO10, BFM+05, Sen08] that can disturb the scheduling of threads, for example, by introducing random sleeps, to further increase the coverage of the interleaving space.

The stress testing approach has several advantages, such as typically requiring a simple setup and having low run-time overheads. However, because these approaches do not have fine control over the interleavings that are explored, the common limitation to these approaches is that they do not systematically explore the thread interleaving space

To address this limitation, a different class of tools has been proposed to test for concurrency bugs [MQB+08, BKMN10, NBMM12a]. This approach relies on taking full control of the scheduling of threads to avoid redundant interleavings and, therefore, increases the effectiveness of testing [BKMN10]. A previous attempt [Blu12] to systematically test kernel code has focused on small-scale educational kernels and relied on modifications to the tested kernels. SKI follows the systematic approach, but distinguishes itself from existing tools by being applicable to kernel code and by being scalable to real-world kernels.

Because the interleaving space is extremely large, systematic tools take advantage of different techniques to restrict the interleaving exploration while still ensuring effectiveness. Examples used in the context of user-mode testing include preemption bounding [MQB+08], reschedule bounding [BKMN10] and the elimination of redundant interleavings [God97]. Other work has proposed limiting the valid run-time schedules by reducing or eliminating the interleavings non-determinism [WTH+12, YCW+14, CWTY10, LCB11, DLCO09, CSL+13, CWG+11]. Restricting the interleavings by applying these techniques could further increase the effectiveness of systematic testing approaches.

Symbolic execution [Kin76, Cla76, CDE08, CKC11] is an analysis technique that systematically explores the application execution path space by keeping track, during execution, of symbolic values instead of concrete values. Symbolic execution has been applied to multi-threaded applications by implementing a custom user-mode scheduler [KZC12]. More recently, SymDrive [RKS12] has been successful at testing kernel device drivers using symbolic execution, although it requires modifications to the kernel and does not target concurrency bugs. Similarly, SWIFT [CRCM12] uses symbolic execution to test kernel file system checkers but does not target concurrency bugs.

Similarly to shared-memory systems, which are the focus of this dissertation, distributed systems are also prone to schedule-dependent bugs [Liu07, LZL+13b, RKW+06] and the complexity of distributed systems also justifies the need for dedicated techniques to scale to real-world applications. For example, CrystalBall [YKKK09] proposes model checking live systems and steering their execution away from states that trigger bugs. By exploring states based on snapshots of live systems, CrystalBall is able to explore states that are more likely to be relevant to the current execution than conducting the entire exploration from a single initial state. MoDist [YCW+09] also finds bugs in distributed systems but does so transparently, without requiring implementations to be written in special languages. MoDist is able to scale to complex implementations by judiciously simulating events that typically trigger bugs, such as the reordering of messages and the expiration of timers.

Input space exploration

Dynamic testing techniques require running the tested software and providing it with testing input. The traditional approach has relied on manually writing test cases [GHK+01], but more sophisticated approaches have been proposed to address this challenge. Such approaches include blackbox fuzzers [BM83], semantically-aware fuzzers [TRI, Ait02] and symbolic execution techniques [GKS05, CDE08, Kin76].

Because file systems have a particularly large input space and are critical components in the system, file system testing has been a particularly active area of research [ADADB+06, YSE06, YTEM04, CRCM12, MDADAD13]. Even though the focus of PIKE and SKI is on the detection of concurrency bugs and on the exploration of the interleaving space, respectively, to evaluate our testing tools, we explored the input space using adapted test suites.

2.3 Deterministic replay

Determinism is valuable for diagnosing concurrency bugs [AWHF10, LCB11, BYLN09, DLCO09, CSL+13, CWG+11, SSV13], but ensuring determinism is orthogonal to the problems that PIKE and SKI address.

Given the same, fixed testing parameters, SKI, like its user-mode counterparts, can deterministically re-execute the same schedule, provided the kernel is given identical input in each run. Currently, SKI does not ensure that the same hardware input is provided to the kernel (e.g., low-granularity timer values). However such guarantee could be provided by SKI by augmenting SKI with a deterministic layer, running below the VMM [LCB11], or by modifying the VMM [DLFC08, XMS+07, SSV13]. In the case of PIKE determinism could be guaranteed by ensuring that the application receives the same input from the operating system on every execution [LVN10, SKAZ04].

2.4 Virtual machine introspection (VMI)

Several VMM mechanisms have been proposed to infer high-level information of virtual machines [CN01]. In many cases the purpose of these mechanisms is to increase performance. Examples include improving the host memory usage by inferring which guest memory is actively being used [CLC13], improving IO performance by anticipating IO requests [JADAD06] and improving the scalability of virtual machine monitors by inferring whether the virtual machine is executing critical sections [WLC+11, ULSD04]. In addition, VMI techniques have been leveraged to gather information about virtual machines in security contexts [NBH08].

Using similar introspection techniques, SKI infers the liveness of threads for the purpose of achieving fine-level control over the threads schedules. For example, SKI leverages the observation that the PAUSE instruction is typically associated with spin-locks, as does the work of Wang et. al [WLC+11] in the context of increasing VMM performance.

3 Background

This chapter provides background on the approach of systematically exploring the interleaving space of software and, specifically, describes a state-of-the algorithm that follows this approach for testing user-mode applications – the PCT algorithm [BKMN10] We leverage PCT, in Chapter 5, to test user-mode applications and we generalize PCT, in Chapter 6, to test kernels.

In addition, this chapter discusses two important types of software, which are analyzed and tested in this dissertation, that are typically particularly complex: database management systems and operating system kernels. Besides being highly complex, both types of software have a critical role in our societies. In fact, databases are fundamental to support the core business functions of many companies and the entire reliability of computer systems typically depends on operating systems. Section 3.2 provides background on database management systems and Section 3.3 provides background on operating system kernels.

3.1 Systematic exploration of the interleaving space

The *systematic* exploration of the interleaving space, in contrast with the stress testing approach, relies on judiciously controlling the thread schedule for each execution of the software under test to maximize the coverage of the interleaving space.

To achieve systematic control over interleavings, these approaches rely on a custom thread scheduler that implements two basic mechanisms:

- **Liveness inference.** The first mechanism infers thread liveness to understand which schedules it can choose. Liveness can be inferred by intercepting and understanding the semantics of the synchronization functions.

- **Schedule enforcement.** The second mechanism overrides the regular scheduler by allowing only a specifically chosen thread to make progress at any point in time. The schedule can be enforced by creating additional scheduling constraints that reduce the freedom of the standard scheduler.

Often, both of these essential mechanisms, are portably implemented through a proxy layer (e.g., through *LD_PRELOAD* or *ptrace*) that intercepts all relevant synchronization primitives to infer and override the liveness state of each thread [MQB+08, BKMN10, NBMM12a].

In addition to the liveness inference and schedule enforcement, the systematic approach also requires a scheduling algorithm to select the schedules explored during each execution. In the following section, we discuss in detail an important algorithm –the PCT algorithm –which is leveraged in this dissertation.

3.1.1 PCT scheduling algorithm

Both PIKE and SKI rely on the PCT scheduling algorithm. As discussed in Section 2.2.2, PCT is part of a class of testing algorithms that systematically explores the interleaving space of user-mode applications. As opposed to stress testing approaches, systematic approaches increase the effectiveness of testing by avoiding redundant interleavings and by prioritizing interleavings that are more likely to expose bugs, e.g., those that differ more from interleavings that have already been explored. PCT has been shown to be more effective, both analytically and empirically, than traditional *ad hoc* approaches [BKMN10].

Algorithm

Conceptually, PCT implements a custom scheduler that executes instructions sequentially one by one; that is, at any point during the execution, only one of the live threads is allowed to progress, and the eligibility of the thread to execute another instruction is re-evaluated after each instruction. Through this process, the scheduler is able to effectively control the chosen interleaving.

To achieve a good diversity of schedules across different runs, the scheduler uses two strategies. The first strategy is to randomly assign initial priorities to the threads, and use these priorities instead of a fixed order to determine the thread that should run at each instant – this is the thread with the highest priority among those that are not blocked.

The second strategy consists of reducing, at random points during the execution of a test, the priority of the thread that is scheduled. If the priority decrease is large enough, this will cause another thread to become the one with the highest priority, and therefore this other thread will be scheduled to run. By varying both the initial priorities and the location of such *reschedule points* in a controlled way, the scheduler is able to control the range of tested schedules.

The reschedule points are chosen prior to each run by randomly selecting a set of offsets from the start of the test (in terms of the total number of instructions executed) within a certain range. Then, during the execution, whenever the total number of instructions executed reaches one of these offsets, the priority of the currently scheduled thread is lowered so that it becomes the lowest-priority thread, and thus another runnable thread is selected for execution in the next step.

The set of reschedule points is determined according to two parameters: the expected number of execution steps k and the desired number of reschedule points p, with the simple interpretation that there will be up to p reschedules within the first k instructions of the execution of the test (and none thereafter, should the test execute for more than k instructions). That is, for a given k and p, the set of p reschedule points is selected by choosing uniformly at random p offsets from the range $[0, k]$.

Effectiveness

Burckhardt et al. define the concept of *bug depth*, in addition to proposing the PCT algorithm. The authors of PCT define the depth of a bug as "the minimum number of scheduling constraints that are sufficient to find the bug" and postulate that, in practice, most concurrency bugs have a small depth [BKMN10]. In fact, the empirical evaluation that the authors have conducted suggests that often concurrency bugs have depths that are limited to 1 or 2.

The PCT algorithm has been shown to be empirically effective, in particular more effective than CHESS [MQB+08], and additionally has the advantage of providing a probabilistic guarantee – the PCT algorithm guarantees that concurrency bugs of a given *bug depth*, p, in the context of single run of a program with n threads and k instructions, are exposed by the scheduler with a probability of at least $1/nk^{p-1}$.

3.2 Database management systems

This dissertation considers MySQL, a widely adopted implementation of a relational database (RDBMS), for the purpose of analyzing real-world concurrency bugs and for evaluating PIKE. MySQL supports the SQL interface, a complex interface, specified by hundreds of pages [SQL], that is the de-facto standard for interacting with RDBMSs. In addition to implementing support for this complex interface, databases often have other important constraints, such as the need to provide fast performance and the need to satisfy the ACID properties, that further increase the challenge of ensuring a correct database implementation. The rest of this section provides background information on MySQL that is relevant for Chapter 4 and Chapter 5.

3.2.1 MySQL overview

MySQL represents an interesting and challenging case study for our work on analyzing and detecting concurrency bugs for several reasons. First, it is a large, complex codebase, with about $360,000$ lines of (mostly C and C++) code and rich application semantics. Second, databases are a critical component of the IT infrastructure of many organizations and therefore it is important to maintain and improve their robustness. In particular, MySQL represents a share of 40% of the database market [MYSc], and is by far the most popular open-source database server. Finally, MySQL is a mature application with a quality development and maintenance process.

3.2.2 Internal structure

MySQL is a complex code base where the state of the server is spread across multiple data structures that are stored both in memory and persistently. In this section, we describe some of the main data structures that will be referred to in later sections.

The *query cache* structure contains pairs of recent instructions that read the state of the database (*SELECT* statements) and their respective results. This structure has been found by its developers to be critical for servers to achieve good performance in many common scenarios. The *query cache*, as one would expect from a cache, should invalidate the relevant entries when they become obsolete due to subsequent and conflicting writes. If the invalidation logic in the application is incorrect it is likely that such mistakes will lead to bugs in which the application returns the wrong results to clients.

The *table cache* stores a set of descriptors, each of which is an in-memory representation of a table schema. When a new thread wants to manipulate a table, it first queries the *table cache* to get a table instance directly if available. Otherwise, in case of a miss, the table schema will be loaded from disk and a new entry will be inserted into the *table cache* structure.

Another important data structure are the *data files.* A *data file* is a critical data structure that stores the actual records for a particular table and is maintained in persistent storage.

To quickly perform searches and find the relevant records in a table, avoiding sequentially scanning the whole table, MySQL also maintains for each table an *index file* which consists of a set of indices. Each entry in the *index file* consists of a pair of elements. The first element is a *key* (or a group of keys) while the second element is a pointer to the appropriate record in the *data file*.

The *key cache* is a repository for frequently used blocks from the *index files* of all tables. The index block will be loaded into the *key cache* before the first access to a table. From that moment on all subsequent operations will be performed on *key cache* data and will be flushed back to disk at the appropriate time.

Finally, the *binary log* (binlog) is another important data structure in MySQL. It stores a sequence of all operations that changed the database state, in their order of execution. This structure is critical for replication. Replicas keep their state in sync by shipping the *binary log* between them and re-executing the requests in the order they appear. Missing entries, wrong entries or entries in the wrong order will likely cause replicas to diverge and therefore it can seriously affect the correctness of the service. Additionally the *binary log* is important for recovery purposes.

3.2.3 Concurrent programming

The use of concurrency in MySQL is typical of a server application. Clients issue several requests to the database server, which are grouped into sessions (called connections). Each connection is handled by a separate thread on the server side, and different threads contend for access to many shared data structures, such as the ones we mentioned above. To synchronize access to these structures, threads mostly resort to locks but also use condition variables.

3.2.4 Request vs. transaction concurrency

To correctly understand the meaning of concurrency bugs the distinction between request and transaction-level concurrency needs to be clear. In a database system, client operations are logically grouped into transactions, each of which consists of a sequence of requests (e.g., requests to begin a transaction, read or write to the database, and commit or abort the transaction). There is often some confusion between the notion of concurrent transactions and concurrent requests, and which types of concurrency bugs are the target.

Both for our concurrency bug study and PIKE we decided to focus on bugs that are triggered by concurrent individual requests, since these are the ones that reflect the traditional concurrency problems that arise in parallel programs. Bugs that are triggered by concurrent transactions but can be reproduced deterministically by a given sequence of requests are not considered concurrency bugs.

Thus we define a concurrency bug as one where the application deviates from the intended behavior, given a certain pattern of inputs, but it must be the case that the bug is only manifested under specific thread interleavings. This definition is general enough to include both safety problems (e.g., server crash or issuing wrong replies) and liveness problems (e.g., deadlocks or even performance bugs).

3.2.5 Storage engines

One of the characteristics of MySQL is that it supports different mechanisms, which are called storage engines, for internally representing and manipulating the state of the database. Users can control which storage engine to use dynamically by parameterizing certain requests during runtime (e.g., *Create Table*) or specifying configuration options set by an administrator. Storage engines represent a significant fraction of the source code of MySQL and implement important parts of the database functionality such as support for indexes and caches, the granularity of locks, and support for compression, replication, or encryption.

One of the most important storage engines in MySQL is the MyISAM storage engine. MyISAM [MYI] is considered to be one of the most popular storage engines of MySQL [STO] and it has also traditionally been the default storage engine [MYI]. In comparison to other engines, MyISAM is optimized for throughput, and is distinctive in that it does not provide the ability to group multiple operations into transactions: instead users have at their disposal explicit locking mechanisms to enforce consistency among groups of operations.

3.3 Operating system kernels

Many important operating system kernels (e.g., Windows, Linux, FreeBSD and OS X) are extremely complex, having millions of lines of source code. This complexity is explained by the large set of functionalities that common-place monolithic kernels are expected to implement and is responsible for making kernels testing particularly challenging. In addition, because kernels operate at the lowest level in the software stack, it is generally harder to instrument the kernel. The rest of this section provides background on kernels that is relevant for the work on SKI (Chapter 6).

3.3.1 Concurrency and synchronization mechanisms

Concurrency arises in the kernel from multiple sources. The first source is the interrupt mechanism. The second source of concurrency arises from in-kernel preemptions, i.e., when processes are preempted while executing system calls or exception handlers in kernel mode. In-kernel preemptions are even more significant in fully preemptible kernels [Lov10] (such as many versions of Linux), where involuntary preemptions may be enacted by interrupts between any two kernel instructions (unless explicitly disabled in short critical sections). Finally, the third major source of concurrency stems from the parallel execution of kernel code on several processors in multiprocessor systems.

To ensure correct semantics, it is possible for kernel developers to restrict all these forms of concurrency, for example, by using locks. In fact, kernel developers typically have at their disposal a large range of synchronization solutions (e.g., per-CPU variables, atomic operations, barriers, spin locks, semaphores, seqlocks, enabling/disabling interrupts, enabling/disabling softirqs). However, restricting concurrency has a negative performance impact, in particular on modern multicore platforms. Kernel developers are thus under significant pressure to increase the supported degree of concurrency, in an effort to prevent the kernel from becoming a performance bottleneck. For example, representative efforts in the past have included switching to locks with finer granularity, adopting new synchronization mechanisms (e.g., RCU [MS98]), and rewriting key internal algorithms of the kernel. Experience shows that such major revamping efforts of concurrent code are rarely bug-free on the first attempt.

Thus, concurrency bugs in kernels are likely to continue to exist, and represent an important threat to the stability and correctness of computer systems in general. We therefore consider it to be important to develop effective tools, such as SKI, that aid developers in finding kernel concurrency bugs.

3.3.2 Thread affinity

SKI leverages the kernel *thread affinity* mechanism, provided by most modern operating systems (e.g. Linux, Windows, MacOS, FreeBSD), to achieve fine-level control of the interleaving of the tested kernel. The thread affinity allows user mode applications to impose scheduling constraints and is commonly used in the context of improving application performance [VZ91]. In our work, SKI leverages thread affinity for testing purposes by imposing a one-to-one mapping between threads, running inside the virtual machine, and CPUs.

4 The effects of concurrency bugs

4.1 Overview

To improve the methods for addressing concurrency bugs, it is important to have a thorough understanding of the characteristics of these bugs. While a few studies of concurrency bugs exist [LPSZ08, FNU03, CC00], they either focus on artificially injected bugs, or, in the few cases where real applications were studied, they mostly focus on the *causes* of these bugs, and limit the study of their effects to whether they cause deadlocks or not. Such studies are useful for determining what kinds of programming mistakes are typical of such applications, and can drive the design of program analysis tools for finding these bugs [PLZ09].

However understanding the *effects* of concurrency bugs is important for a different set of reasons than why it is interesting to study their causes. Analyzing the effects allows us to assess how efficiently existing detection approaches handle these bugs. And, more importantly, it can serve as a guide for further development not only of tools and methodologies that detect, but also of tools and methodologies designed to tolerate and recover from the faults and errors caused by such bugs. To give a simple example, it is important to understand how often concurrency bugs cause failure modes where the server returns incorrect replies, as opposed to not providing a reply at all, in order to gauge the effectiveness of using multi-threaded replicas to ensure fault diversity in a Byzantine-fault-tolerant replication scheme [CL99].

In this part of the proposal we provide the complementary angle of studying the *effects* of concurrency bugs that affect parallel applications. In particular, we exhaustively study real concurrency bugs that were found in MySQL [MYSa], a mature, widely-used database server application.

Our study produced several interesting findings. First, we found a non-negligible number of *latent concurrency bugs*. Latent concurrency bugs, when triggered, do not become immediately visible to users. Instead, these concurrency bugs first silently corrupt internal data structures, and only potentially much later cause an application failure to become externally visible[1]. Latent concurrency bugs have been anecdotally reported [EA03], but we are the first to study their extent, and their internal and external effects in detail.

A second finding is related to bugs that cause the application to fail in ways other than silently crashing – semantic concurrency bugs. Some of our findings were surprising, like the fact that these bugs cause subtle changes in the output that would be difficult to find using existing run-time monitoring tools, or the fact that there exists a strong correlation between semantic bugs and latent bugs.

Our findings have implications for the design of tools and methodologies that address concurrency bugs. For the convenience of the reader we present a summary of our main findings together with their implications in Table 4.1.

The remainder of this Chapter is organized as follows. In Section 4.2 we describe our methodology. The results of our study are presented in Section 4.3 and in Section 4.4 we discuss their implications. We present a summary of this Chapter in Section 4.5.

4.2 Study methodology

In this section we present the methodology that we adopted to find and analyze concurrency bugs – our methodology is similar to one used in previous work [LPSZ08, ZAH11, JSS+12, NJT13].

We decided to study concurrency bugs in MySQL, because it is a critical component for the industry and because it is also a mature, open-source and highly concurrent application. Focusing on one important application allow us to study the concurrency bugs within this scope in greater depth, although it also requires greater care in generalizing the results, as we discuss in Section 4.4.

4.2.1 Selection of concurrency bugs

The MySQL versions that are affected by the bugs that were reported in the bug report database range from version 3.x to 6.x and the oldest bug reports date back to 2003.

The MySQL bug report database contains a very large number of bugs. Therefore, to make the task feasible, we automatically filtered bugs that are not likely to be relevant by performing a search query on the bug report database.

Our search query filtered bugs based on (1) the keywords contained in the bug description, (2) the status of the bug and (3) the bug category.

[1]The term *latent bug* is used in other work [BE04, KWLM09, HP04] with an unrelated meaning – that of a bug that went undetected by the *programmer*.

Finding	**Implication**
Evolution of concurrency bugs	
According to the opening dates of our sampled bugs, the proportion of fixed bugs that involved concurrency more than doubled over the last 6 years.	This suggests an increasing need for new tools and methodologies to handle concurrency bugs.
External effects of concurrency bugs	
We found slightly more non-deadlock bugs (63%) than deadlock bugs (40%).	Having good tools to handle deadlock bugs is not enough – non-deadlock bugs also need to handled.
We found a significant fraction of semantic bugs (15%).	Byzantine-fault-tolerance (BFT) techniques can potentially handle a considerable fraction of concurrency bugs.
Immediacy of effects	
Latent concurrency bugs were also found in significant numbers (15%).	Tools and methodologies such as proactive recovery can be leveraged to mask errors caused by a significant numbers of concurrency bugs.
Of the latent concurrency bugs analyzed, 92% were semantic bugs and conversely 92% of the semantic bugs were also latent bugs.	Given the high correlation between these classes of bugs, techniques that handle one class should also handle the other.
Semantic concurrency bugs	
The vast majority of semantic bugs (92%) generated subtle violations of application semantics.	Run-time monitoring and testing tools will have to devise complex application-specific checks to detect the presence of semantic bugs.
Internal data structures	
Most of the examined latent bugs (92%) corrupted multiple data structures.	Techniques that detect inconsistencies among data structures could be used to detect latent bugs. Analyzing data structures individually might not suffice.
Severity and fixing complexity of bugs	
Latent bugs were found to be slightly more severe than non-latent bugs.	Latent bugs are an important threat to software reliability and, therefore, latent bugs should also be addressed.
Latent bugs were found to be easier to fix than non-latent bugs.	Further studies should be performed to analyze the reasons for this difference.

Table 4.1: Main findings of this study and their implications. The methodology for collecting the data presented here is described in Section 4.2 and the results are explained in detail in Section 4.3.

We searched the MySQL bug report database for bug reports with descriptions containing keywords commonly associated with concurrency bugs (Table 4.2). In addition to searching by keywords, we searched for bugs whose status was *closed* (i.e., bugs that are no longer under analysis by the developers/debuggers). It would have been interesting to also consider bugs with other status (such as *won't fix* and *can't repeat*) but these

Keywords
atomic
compete
concurrency
deadlock
lock
mutex
race
synchronization

Table 4.2: Concurrency-related keywords used for selecting bug reports.

Phase	Number of bugs
Total MySQL server closed bugs	12.5k
Concurrency related keyword matches	583
Sampled bugs	347
Concurrency bugs analyzed	80

Table 4.3: Bug counts for different stages of the analysis.

bug reports are not likely to have detailed discussions and more importantly, in general, they will not contain patches. Without reasonably complete bug reports it would not be possible to thoroughly understand the bugs they report.

Next, to exclude bugs from stand-alone utilities that are unrelated to the multi-threaded server, our search query also limited the search to bugs that were related to MySQL Server, including those that were within the Storage Engines category [Pac07].

Finally, we randomly sampled a subset of the bugs that matched our search query and manually analyzed them. The manual inspection revealed that some of the bugs that matched the search query were not concurrency bugs and so we also excluded them. In addition, we excluded bug reports that did not contain enough information to analyze them. After filtering, we obtained a final set with 80 concurrency bugs that were analyzed – a number that is very close (or even superior) to the number of bugs analyzed in previous concurrency bug studies [LPSZ08, CC00]. Table 4.3 shows the bug count across the different stages of the bug selection process.

Note that the selection process used has two main limitations. First, the search query can miss some actual concurrency bugs. However, a concurrency bug report that does not contain any of the main keywords associated with concurrency is also more likely to be incomplete and therefore more difficult to successfully analyze. Second, concurrency bugs are likely to be underreported, which would explain why out of a total of about 12.5k bugs in the bug database we only found 80 concurrency bugs.

4.2.2 Manual analysis of bug reports

We manually analyzed the bug reports of the sampled list of bugs, focusing on trying to understand the effects of the bugs. We analyzed the bugs using information contained in the bug reports (including the patches), as well as the source code of the application.

Bug reports contain several types of information that are useful for filtering out non-concurrency bugs, and for understanding their characteristics. In particular, bug reports contain not only the description of the bug, but also discussion among the developers and users about how to diagnose and solve the problem. The information contained in these discussions is often important to understand the bugs, in particular to determine whether they are concurrency bugs, and to understand their effects. Typically the bug report will also include the patch, and even the method to reproduce the bug; sometimes more than one patch attempt is made before developers agree on a definitive patch. Bug reports also include additional fields such as the perceived severity, the status, and the software version affected.

We used all these types of information contained in bug reports to gain an understanding of how bugs are triggered and their effects[2]. In addition, some of this information was also used to estimate the complexity of fixing concurrency bugs and their severity.

4.3 Results

In this section we present the results of our analysis of the 80 concurrency bugs that we found in the MySQL bug database. A summary of these results and their main implications are also presented in Table 4.1.

4.3.1 Evolution of concurrency bugs

We investigated the proportion of concurrency bugs present in the bug database and how this proportion evolves. We were interested in knowing whether concurrency bugs are becoming more prevalent. To determine this, we identified the opening and closing year of the concurrency bugs that we analyzed as well as of all closed bugs within the MySQL server category. To obtain the set containing all bugs we excluded the keyword part of the search together with the sampling phase explained in Section 4.2. For each year we counted the number of concurrency bugs and their proportion (compared with generic bugs). We looked at both the opening date and closing date because programmers

[2]The raw data gathered from this manual analysis can be found at http://www.mpi-sws.org/~pfonseca/dsn2010-bug-study.tgz

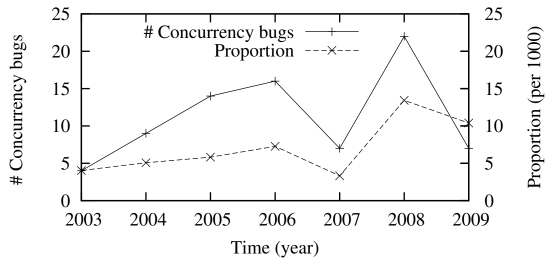


Figure 4.1: Evolution of bugs (by open date).

typically require a significant amount of time (i.e., many months) to solve the bugs under analysis. The results are presented in Figures 4.1 and 4.2. From these results we can see that there has been a trend of increasing number and proportion of concurrency bugs over the years. However, this trend does not seem to be very prominent.

The data that we collected does not allow us to determine the causes underlying this finding, however we can think of two possible reasons for this slight increase. One possible explanation is that the advent of multi-core hardware causes users and developers to stumble upon these bugs more often than they used to in the past. Another explanation that we cannot rule out is that developers, while trying to further parallelize the code, actually increase the number of concurrency bugs that they introduce.

Of the concurrency bugs that we sampled, the oldest concurrency bug was opened in March 2nd, 2003, while the most recent was closed in September 16th, 2009. Therefore, to make the comparison fair, we excluded the bugs that were outside this range from the list of generic bugs used to compute the proportions.

To interpret these results it should also be taken into consideration that, as we show in Section 4.3.7, the time it takes to close a concurrency bug can be quite long (e.g., some bugs took more than a year to fix). This explains why the absolute number of bugs opened in the last year is low: many concurrency bugs potentially discovered in 2009 had not been fixed at the time our analysis took place (early 2010), which means they were not yet closed and were, therefore, not accounted for in this study.

4.3.2 External effects

We analyzed the concurrency bugs with respect to the external effects that are exposed to the clients, and divided these effects into six categories. The results are presented in Table 4.4. Note that the sum of all occurrences is larger than the total number of bugs because some bugs fit into more than one category.

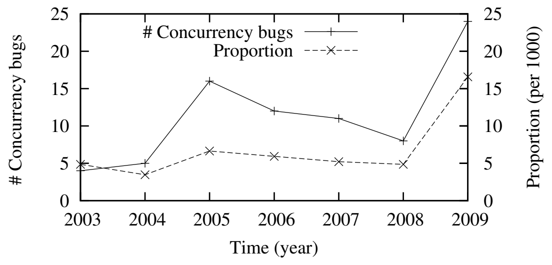


Figure 4.2: Evolution of bugs (by close date).

We can see that there are slightly more bugs that cause non-deadlock conditions (63%) than deadlock conditions (40%), and among the non-deadlock bugs the most prevalent consequences are either causing the server to crash (28%) or providing the wrong results to the user, which we term *semantic bugs* (15%).

Semantic bugs cause applications to provide users with results that violate the intended semantics of the application. This is an interesting class of bugs since masking their effects requires sophisticated (and possibly expensive) techniques such as Byzantine-fault-tolerant replication [CL99] or run-time verification of the behavior of the application against a specification of the system [Sch95]. We discuss these bugs in more detail in Section 4.3.4.

The high percentage of deadlock bugs that we encountered leads us to believe that, despite significant research to address deadlock bugs, in practice this class of bugs still constitutes a significant problem for the robustness of software. The percentage of deadlock bugs that our study found is in line with results from other studies. For example, Lu et al. [LPSZ08] found that up to 30% of the concurrency bugs analyzed were deadlock bugs.

The remaining three classes of external effects were slightly less prevalent. These are error messages (9%), which we distinguish from the class of semantic bugs, despite the fact that when error messages are provided to the user an unexpected result is also returned. We distinguish error bugs from semantic bugs by the fact that an error is detected by the server and, therefore, is explicitly flagged in the reply to the client request, and can be handled by the client application appropriately. For instance, in one bug (bug #42519) when a restore operation is performed concurrently with an insert operation a generic error message is returned to the user. We also found a number of bugs (8%) in which client requests hang (the client does not receive a reply), which differs from a deadlock situation where one thread or a series of threads are waiting in a circular dependency. Typically, non-deadlock bugs are caused by a thread that fails

External effect	Number of bugs
Crash	22
Deadlock	32
Error	7
Hang	6
Performance	5
Semantic	12

Table 4.4: External effects of concurrency bugs.

to release a certain lock, causing another thread that tries to acquire it to wait forever. Finally, we found a few (6%) concurrency bugs that caused performance degradation (e.g., memory leaks that increase the number of page faults the server incurs).

4.3.3 Latent bugs

Next we analyzed whether the bugs caused latent errors or not. We define a latent bug as one where the (concurrent) requests that cause the erroneous state to occur differ from the request (or requests) that cause the external effects of the bug to be exposed to the clients (i.e., the violation to the application's specification). In other words, latent bugs cause internal data structures to be silently corrupted (i.e., an error) but do not immediately cause a wrong output (i.e., a failure). A failure is only triggered by a subsequent request that may not have to run concurrently with any other requests.

We found that a relevant fraction of concurrency bugs in our study were latent (15% versus 85% non-latent bugs). This result was somewhat surprising and has an interesting implication. The fraction is large enough that we believe there is value in developing tools that try to recover the internal state of the concurrent application. Performing such a recovery could prevent concurrency bugs from affecting the correct behavior of the application, even after the concurrent requests that cause the error have already been executed and the application state is corrupt.

We also analyzed how latent bugs were categorized according to the previous analysis of their external effects. The results in Table 4.5 show a very high correlation between latent and semantic bugs: 92% of the latent bugs manifest themselves by returning wrong results to the client, and conversely also 92% of the semantic bugs are latent. (The fact that these values are exactly the same is only a consequence of the relatively small sample size.)

We see two possible consequences of the high correlation between latent and semantic bugs. On the one hand, methods to address the problems caused by latent bugs will have to take into account that they manifest themselves through violations of the application semantics (rather than crashing or halting), which raises the bar for detecting when a latent error is activated and becomes a failure. On the other hand, this opens an opportunity for the methods that handle non-crash faults to try to heal the state of the application in the background instead of masking the effects of these faults in the foreground. For instance, rather than tolerating semantic errors using Byzantine-fault-tolerance (BFT) replication, where the output of each request is voted upon, one might be able to get similar results by having a foreground replica that issues the reply, and a background replica that checks and recovers the service state.

A concrete example of a latent bug will help the reader understand some of the typical patterns surrounding bugs that are both latent and semantic. Bug #14262 involved concurrent requests updating both the contents of the database (e.g., table contents) and the binlog structure. This bug is caused by the code not enforcing the same order for concurrent requests that update both the table contents and the binlog. Thus, when a specific set of statements is sent to the primary replica, the primary replica updates the table data by executing the statements in one order but, depending on the exact interleaving of threads, may write those statements to the binlog in the reverse order. The result of this bug to the client is only visible after a fault of the primary replica occurs (or when clients otherwise contact the backup replicas). In this case, one of the backups will take over with a state that diverges from the previously observed state (in that it reflects a different sequence for transaction execution) and subsequent results will be incoherent with those that were previously returned.

In the remainder of this section we will analyze semantic and latent bugs in more detail. The reason for our focus is twofold. First, we found these bugs to have a relevant (and perhaps unexpected) prevalence. Second, and more importantly, although existing tools are very effective at handling application crashes (e.g., Rx [QTSZ05]) and deadlocks (e.g., Dimmunix [JTZC08]), they are not so effective at handling the remaining, more subtle types of failures. Thus, there is a research opportunity for improving methods that address this type of concurrency bug.

4.3.4 Characteristics of semantic bugs

We further analyzed the incorrect outputs returned by semantic bugs in order to determine how difficult it is to detect them, e.g., using a run-time monitoring tool [Sch95], which would avoid the use of more expensive techniques such as BFT replication [CL99].

Out of all the semantic bugs, we found only one to have a self-inconsistent output, meaning that the buggy output clearly deviated from the expected reply. In this partic-

External effect	Number of bugs
Crash	1
Deadlock	0
Error	0
Hang	0
Performance	1
Semantic	11

Table 4.5: Effects of latent concurrency bugs.

ular bug, the wrong reply returned to clients contains information about the contents of a certain table, but at the same time the reply also contains information that indicates that the table does not exist in the database.

None of the remaining bugs were self-inconsistent, implying that there are limited benefits from detection techniques that try to validate the correctness of the application by analyzing the replies.

We further analyzed these results and categorized the output of semantic bugs into two groups. Some of the bugs did not fit into either of these groups.

The first group, containing 58% of these bugs, corresponds to outputs that reflect an ordering of previously executed transactions that is inconsistent with the ordering that was implied in previous replies. The latent bug we described before where binlog entries were logged in the wrong order is an example of such a bug: after the primary becomes faulty, the output of the system reflects the order in which transactions were recorded in the binlog, which differs from the order in which they had been originally executed.

The second group, containing 25% of the bugs, corresponds to violations of transactional semantics, in particular of the isolation property of the transactions. This means that transaction A could see the intermediate effects of a concurrent transaction B (e.g., some of the updates made by transaction B, but not all of them).

Finally, 17% of the semantics bugs did not fall into either of the previous two categories.

4.3.5 Internal effects of latent bugs

We also analyzed the set of latent bugs in more detail. In our analysis, we paid close attention to how the internal state was being corrupted, so that we could gain better understanding of the kinds of techniques that can be useful for detecting the errors before they are exposed to the user and for recovering the internal state of the application.

Data structure	Number of bugs	Persistent?
Data file	11	Yes
Index file	9	Yes
Definition file	8	Yes
Query cache	7	No
Key cache	6	No *
Binlog	5	Yes

Table 4.6: Most frequent data structures involved in latent bugs. * The contents of this cache can also be written back to disk.

First, we determined whether each bug corrupted a single high-level data structure, or modified two or more data structures in an inconsistent way (leaving them in an incorrect state relative to each other). Only 8% of the latent concurrency bugs involve a single data structure, and the remaining 92% involve inconsistency between separate structures.

Next we analyzed whether the data structures involved are persistent structures stored on disk or volatile structures kept in memory. Table 4.6 shows that the three most affected data structures are persistent, namely the files that contain the database contents, the respective indices, and the aforementioned binlog file. We also found a large number of bugs involving caches that are only stored in main memory.

Note, however, that these results do not allow us to draw conclusions about the probability that accesses to these data structures trigger bugs, given that we do not know how often different structures are accessed (and also we cannot claim that we have a perfectly representative sample of the existing bugs).

Note that the numbers in Table 4.6 do not add up to the total number of latent bugs because certain bugs affected more than one data structure, as explained before.

4.3.6 Recovering from latent errors

We looked at the ability of the application to recover from latent bugs after they have caused an error (i.e., corrupted the internal state). The recovery mechanisms we consider in this section are relatively simple ones: we identified the latent errors that can be recovered by a server restart or other simple mechanisms (e.g., reloading indexes) that do not require writing extensive recovery-specific code. We present the results in Table 4.7. Note that some bugs allow more than one simple recovery mechanism.

We found that in one third of the cases it is possible to use simple mechanisms to recover latent errors such that they go completely unnoticed by users. This increases the chances of adopting proactive recovery techniques.

	Number of bugs
No simple recovery mechanism	8
Allow for simple recovery:	4
Server restart	4
Other mechanisms	3

Table 4.7: Recovery mechanisms for latent concurrency bugs.

4.3.7 Severity and fixing complexity

Finally, we compared concurrency bugs belonging to different categories with respect to their severity and to the complexity of fixing them, according to the bug report fields that specify these properties. Additionally, we also compared non-latent bugs against latent bugs with respect to these two properties.

The average severity of bugs is compared in Table 4.8. The results show that latent bugs were considered to be slightly more severe on average than non-latent bugs. In the ranking of severity by external effects, crash bugs were found to be the most severe while, as expected, performance bugs were found to be the least severe.

For the complexity of fixing concurrency bugs we used four metrics that we extracted from the bug reports: time to fix the bug, number of patching attempts, number of files changed in the final patch, and the number of comments exchanged in the bug reports. Although none of these metrics is perfect, in combination they help us estimate the complexity of fixing these bugs. We present a comparison of the four complexity metrics in Table 4.9. Since some of these fields contain significant outliers, in addition to presenting the average for all four metrics we also present the median.

Our analysis of the fixing complexity revealed a surprising result: non-latent bugs were found to be more complex to fix than latent bugs in all metrics except for the number of patches. We currently do not have a clear explanation for this fact.

4.4 Limitations and discussion

One of the results of our study is that the percentage of concurrency bugs present in the bug database is low. This is not very surprising, since it has long been believed that concurrency bugs are underrepresented. The fact that concurrency bugs are hard to observe and reproduce (in fact they are commonly referred to as Heisenbugs [Gra86]) is likely to contribute to their underrepresentation in bug databases for three main reasons. First, 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 at all. Second, even when users

Bug immediacy	Severity
Latent	2
Non-latent	2.2

Bug category	Severity
Deadlock	2.3
Crash	1.7
Error	2.4
Hang	2
Performance	3
Semantic	2.2

Table 4.8: Average severity of concurrency bugs according to their immediacy and category. Maximum severity is rated as 1 (i.e., critical bug) while minimum severity is rated as 5.

Bug immediacy	Time	Patches	Files	Discussion
Latent	114/79	3.8/2	2.3/1	10.4/7.5
Non-latent	137/90	2.7/2	3.9/1	11.6/9

Bug category	Time	Patches	Files	Discussion
Deadlock	125/90	1.9/2	1.5/1	9.3/9
Crash	128/83	3.5/2	7.7/3	12.9/11
Error	150/94	3.0/2	4.4/4	17.0/11
Hang	210/116	4.5/2	3.8/2	13.2/11
Performance	125/92	1.4/2.5	1.8/2	8.2/6
Semantic	108/67	3.8/2	2.2/1	10.5/8

Table 4.9: Complexity of fixing concurrency bugs according to their immediacy and category. For each class of bugs we present the average/median for each of the four metrics: time in days, number of patches, number of files in the patches and the number of comments in the discussion.

are able to reproduce bugs on their machines, it might not be possible to reproduce the bug in the developer's environment due to small differences in the environments. Third, even if developers manage to reproduce the bug, they might not be able to systematically reproduce it using traditional debugging methods, since some debugging tools and methods might interfere with the reproducibility of the bug.

In this study we focused our attention on concurrency bugs found in the MySQL application. A previous study compared concurrency and non-concurrency bugs of three different database systems including MySQL [VBLM07] and it concluded that the three different database systems exhibited a very similar proportion of crash vs. non-crash faults (i.e., a bit over half of the bugs led to non-crash faults in each database system).

While not conclusive, this observation leads us to believe that the bug patterns we found in MySQL might also apply to other database systems. More analyses are required to confirm whether this is in fact the case.

On the other hand, it seems less likely that these results can be generalized to arbitrary multi-threaded applications. Applications can be very different (e.g., some have graphical user interfaces while others do not, some applications use the client-server model while others do not). As an example, from the data collected in another study [LPSZ08] that compared different applications, about half of the deadlocks found in MySQL involved the synchronization of accesses to only one variable while almost all of the deadlocks found in Mozilla involved two or more variables. Given the very different characteristics of applications, we believe that the conclusions that we present here are unlikely to be generalizable to arbitrary multi-threaded applications.

The number of bugs analyzed in this study is comparable to the number of bugs analyzed in other related studies – Lu et al. studied 105 concurrency bugs [LPSZ08], Sahoo et al. studied 30 concurrency bugs [SCA10] and, more recently, Lin et al. studied 90 concurrency bugs [TLL+14]. However, it is worth noting that our results could potentially suffer from two sources of bias. First, our sample, in absolute terms, is small. Obviously, this limits the confidence in the results, but at the same time it is a limitation that is difficult to overcome due to the time required to gather the data and the amount of data available. (This is a limitation shared by previous studies.) Second, we only analyzed bugs that were documented and fixed. This means we did not account for bugs that were not fixed (or even found), nor bugs that were fixed but not documented. We believe that these biases are very difficult to overcome given the nature of bugs in general but specifically given the nature of concurrency bugs. Nevertheless, more studies are desirable to improve our understanding of concurrency bugs.

4.5 Summary

To gain a better understanding of real-world concurrency bugs, this chapter presented a study of concurrency bugs in MySQL. In contrast to previous studies, our study focused on the effects of concurrency bugs rather than on their causes, which is an important aspect for the development of bug detection tools.

Studying how bugs manifest enabled us to reach some important findings, such as the fact that there exists a high prevalence of latent bugs and there exists a strong correlation between latent bugs and semantic bugs that cause silent failures. These findings motivated our work presented in Chapter 5, which aims at detecting automatically either of these classes of bugs in complex applications.

5 Detecting latent and semantic bugs

5.1 Overview

In this part of the work we propose a new technique for detecting latent and/or semantic concurrency bugs that does not rely on data race detectors or on assertions. Our thesis is that it is possible to implicitly extract a specification, even for large multi-threaded server applications, by testing if the application obeys linearizable semantics [HW90]. Intuitively, linearizability means that concurrent requests behave as if they were executed serially, in some order that is consistent with the real-time ordering of the invocations and replies to the requests. While similar ideas have been applied to the design of tools for testing concurrency bugs, they have been limited to testing the atomicity of small sections of the program or library functions with at most hundreds of lines of code [BDMT10, XBH05a, Vaf10]. We push this idea to an extreme by postulating that even a complex multi-threaded server with hundreds of thousands of lines of code can come close to obeying linearizable semantics.

By systematically testing if linearizability is upheld, we can find subtle violations of the application semantics without having to write a specification for each concurrent application. Furthermore, by checking if both the output and the internal state of the application obey the inferred semantics, we can identify not only the bugs that manifest themselves immediately as a wrong output, but also those that silently corrupt internal state. However, achieving a meaningful state comparison requires abstracting away many of the low-level details of the state representation. We accomplish this by means of simple annotations that are provided by the tester. This approach also allows the tester to progressively increase the chances of finding latent concurrency bugs by incrementally annotating the state.

We implemented PIKE, a testing tool that brings together these principles and state of the art techniques for the systematic exploration of thread interleavings. We describe the design and implementation of PIKE, and our experience in applying it to MySQL.

Our experience demonstrates that, despite the size and complexity of MySQL, in practice the semantics it provides are sufficiently similar to linearizability for our detector to be effective. Although we used only a simple battery of inputs for testing (based on the testing inputs that shipped with the application) we were able to find a considerable number of concurrency bugs in a stable version of the database. Furthermore, the effort to provide the required annotations was small, and after installing simple filters we also found the number of false positives to be modest. All of this was achieved without having to figure out which were the correct outputs (or final states) for any given inputs, since PIKE automatically extracts a specification by comparing the outputs and states of different interleavings.

The remainder of this chapter is organized as follows. Section 5.2 presents the problem and gives an overview of our approach. In Section 5.3 we introduce PIKE, the tool that we built to find concurrency bugs. Section 5.4 describes our experience applying PIKE to MySQL. Section 5.5 presents the results that we obtained from our experience and Section 5.7 provides a summary of this chapter.

5.2 Semantic and latent concurrency bugs

This section gives an overview of our main insight to automatically extract the application specification and it explains our approach to analyze the application state.

5.2.1 Linearizability: The spec from within

For testing tools to detect semantic or latent bugs, they need some form of specification for the expected output and state obtained after running each test, in order to determine if a given test run uncovered a bug or not. To address the absence of a manually written specification capturing deviations from the intended application semantics, we propose extracting such a specification from the behavior of the same application but under different conditions.

In particular, our hypothesis is that, even in the case of a complex server application with hundreds of thousands of lines of code, the semantics that are intended by the programmer are normally close enough to linearizability [HW90] that we can use it as a good first approximation of a specification.

To formally define linearizability [HW90], we must first define the notion of history, which is a finite sequence of events that can be either invocation of operations or responses to operations. A history is classified as sequential if its first event is an invocation, each invocation is immediately followed by a matching response, and each response is followed by an invocation. Two histories H and H' are defined as equivalent if, for every process P, the sequence of invocations and responses performed by P is the same, i.e., $H|P = H'|P$. A history H induces an irreflexive partial order $<_H$ on operations such that $o_0 <_H o_1$ if the response event for o_0 precedes the invocation of o_1. Given these definitions, a history of events in a concurrent system H is linearizable if there is an equivalent sequential history S (called a linearization of H) such that $<_H \subseteq <_S$.

Intuitively, this means that, despite its internal concurrency, the server behaves as if requests were processed in sequence, and that this processing took place instantaneously some time between the moment when the client invoked the request and received the respective reply.

Therefore, assuming the application tries to follow linearizable semantics, a testing methodology can be devised by comparing each concurrent execution of the application with all possible linearizations (i.e., all possible sequential executions of the requests) for the same input. If none of the linearizations matches the behavior of the concurrent execution, then a concurrency bug is suspected to have been triggered and an error is flagged.

Testing for linearizability would only require us to inspect the outputs of the concurrent execution against the outputs of the linearizations. This would be sufficient to capture semantic bugs, but not to capture latent bugs. To handle latent concurrency bugs, we can resort to the same principle of testing for linearizability but applying it to the state of the application. This testing methodology is summarized in Figure 5.1. It shows two concurrent requests, R_1 and R_2, whose execution overlaps in time (Conc C). To check if the concurrent execution is linearizable we must compare it to all possible linearizations, namely R_1 followed by R_2 (Seq A) and R_2 followed by R_1 (Seq B). Linearizability is obeyed when both the state and the output of the concurrent execution match both components in at least one of the two linearizations.

Note that by using linearizability as a specification, we are not necessarily extracting a correct specification of the system, not only because the programmer might not have intended the application to obey linearizable semantics, but also because the sequential execution may be buggy, and consequently the deviation to the expected behavior could go undetected. The latter issue is not problematic in the case of concurrency bugs, though, since these arise from the lack of proper synchronization among multiple threads, which does not arise when executing requests without concurrency.

5.2.2 Capturing application state

As we mentioned, to be able to find latent bugs we need to compare both the output and the state of different executions of the application.

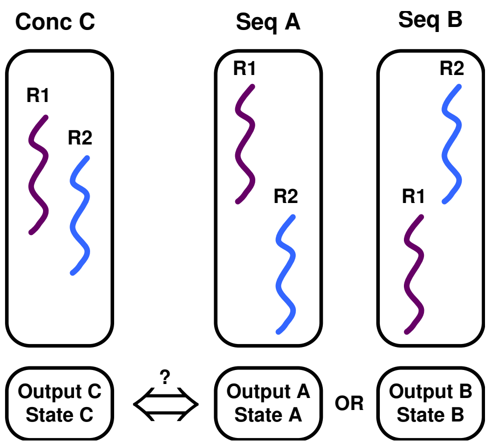


Figure 5.1: Checking for linearizability of state and outputs of two concurrent requests.

While outputs are fairly straightforward to compare, the same cannot be said about the state of the application. In particular, the naïve approach of simply comparing the state of the various executions bit-by-bit is doomed to fail. The reason is that by changing thread interleavings, the low-level state of the executions will quickly diverge. For instance, if we consider operations such as dynamic memory allocation, slight changes in the thread interleaving could easily change the relative order of allocation requests, and therefore the memory layout of allocated heap space would likely be different as well.

We address this by asking the tester or the developer of the application to provide a *state summary function* which captures an abstract notion of the state in a way that takes into consideration the semantics of the state and allows for a logical comparison, instead of a low-level physical comparison. As an example, a data structure that represents a set of elements should be compared across different executions in such a way that is not only oblivious to the memory layout, but, given that sets can be stored in data structures that imply an ordering such as a list, but the order in which the elements of a set are listed is irrelevant, the state summary function must be oblivious to this order.

While writing this extra code could be a burden for the tester or the developer, we found that in practice these functions are simple to write, in part, because the internal interface of the application we analyzed is reasonably well defined. Additionally, we provide a small library that assists programmers in writing state summary functions for the most common types of data structures. Finally, we note that in our testing framework the state summary functions will always be scheduled without preemptions, given our custom scheduler (Section 5.3), and, therefore, do not have to be synchronized with respect to the existing application code.

5.2.3 Maintaining the summary functions

Annotating the application undoubtedly requires some effort from testers. During the life-cycle of the application it might not suffice to annotate it once – it might be necessary for testers to revise the annotations when there are new versions of the application. Major updates to the application (which typically involve substantial code rewrites) are likely to require some effort to update the summaries. However, in practice, we expect that many upgrades to the application will maintain most of the properties of the data structures as well as the interface that is used to access them. In these cases, no changes to the annotations would be required.

5.3 PIKE: A concurrency bug finding tool

In this section we describe how we combine our linearization approach, which analyzes both the output and the state of different interleavings for linearizability violations, with state of the art testing techniques. The result is a bug finding tool geared towards finding concurrency bugs that are traditionally hard to detect.

5.3.1 Handling false positives

One of the challenges we expected to face when deploying PIKE is that linearizability would not necessarily hold for a large, complex application with rich semantics and hundreds of thousands of lines of code. These cases, if not appropriately dealt with, could lead to the tool outputting a large number of false positives.

An example of a data structure that we found to sometimes not obey linearizability is an application-level cache. In particular, this occurred in situations where the application logic detected that two requests were being handled concurrently and that would cause a cache entry that one of them would create to be invalidated. In these cases, the application would conservatively not insert that entry into the cache. This behavior might have an impact on performance but does not affect correctness, i.e., an application can always choose not to insert an entry into the cache. However, if the application were to execute the same requests sequentially, because no possible conflict would exist, the last request would be inserted into the cache.

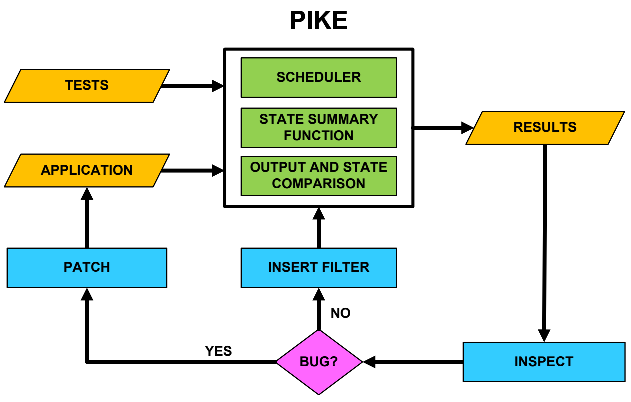


Figure 5.2: Overall architecture of PIKE. The system receives as inputs a multi-threaded application and a test suite, and contains a feedback loop that can be used by testers to insert filters to avoid false positives when the application deliberately violates linearizability.

To handle these cases, the state summary functions break the state up into separate components; e.g., an application-level cache would be an individual component. Furthermore, we allow the tester to write a rule that enables the linearizability test to check for inclusion, instead of equality, among the set of entries in some of the state components. In the case of the application-level cache, this rule might allow for checking whether the set of elements in the cache for the concurrent execution are contained in set of elements in the cache for at least one of the sequential executions. We found this approach to work well, in practice, in reducing the number of false positives to a reasonable level.

Therefore, our final system design contains a feedback loop where testers can add rules that describe such exceptions to linearizability, thus avoiding most false positives and making the problem tractable.

Figure 5.2 illustrates the overall process. Developers provide PIKE with the application and the testing inputs. PIKE will then run the application multiple times exploring different thread interleavings and checking for linearizability of both state and output. To conclude whether a bug was found, the developer then inspects the results produced by PIKE which include the output, the state and information about the interleaving of the various executions. In case the developer finds various cases of similar false positives he can simply insert a rule to adjust the comparison functions and re-run PIKE.

5.3.2 Implementation

As Figure 5.2 also shows, the implementation of PIKE is composed of three components: the scheduler, the *state summary function*, and the component to compare the state and output of the application.

PIKE combines our proposed linearizability detector with a custom scheduler that implements the PCT algorithm (Section 3.1.1) proposed by Burckhardt et al. [BKMN10]. The scheduler enables PIKE to effectively explore the interleaving space during testing. We implemented the scheduler in about $3,000$ lines of C code. Our scheduler controls the thread interleaving by intercepting the library calls of the target application and forcing a single thread to run at a time which is randomly chosen according to the PCT algorithm.

Our scheduler takes control over the application using the *LD_PRELOAD* environment variable and intercepts the pthread library calls made by the application; i.e, the scheduling granularity is at the level of the pthread library calls. Similar levels of granularity have previously been found to produce good results at finding concurrency bugs [MQB+08].

We require application writers to identify the location where the handling code of each request begins and ends. The scheduler needs to know about these locations to force sequential interleavings, i.e., interleavings that execute each request without preemptions from other concurrent requests. This information also helps in debugging the application when bugs are flagged. Since our scheduler only takes control of the application when it makes pthread calls, it could happen that the running thread (i.e., the runnable thread with highest priority) invokes a system call that does not return. In such a situation, the entire application would block – the highest priority thread would be blocked on a system call and the other threads would have previously been blocked by the scheduler. A situation where this would occur is in the location where the main thread of MySQL spawns new threads to handle new client sessions. To avoid this, we make the scheduler aware of that particular location in the MySQL code and make the scheduler block the main thread as soon as it creates all the expected client-session threads (which is dependent on the input). In comparison with the effort to annotate the application for the purpose of capturing the application state, the effort required to identify these three locations was negligible.

The PCT scheduler algorithm requires the definition of variable k and p, as discussed in Section 3.1.1. In our experiments we used the value $50,000$ for variable k, the maximum number of execution steps per run (after the database initialization phase), and we used

$p = 1$, i.e., a single reschedule point. We found empirically that these values produced good results for the application we studied.

The PCT scheduler algorithm also requires an anti-starvation mechanism. Without this mechanism if the highest priority thread enters a busy wait cycle it would never relinquish the processor and would prevent the entire application from progressing. Examples where such situations could occur are the instances where ad-hoc synchronization methods are used [XPZ+10]. We implemented the anti-starvation mechanism simply by reducing the priority of the running thread if it runs uninterrupted for more than a certain number of execution steps. We found this mechanism to be particularly useful during initialization periods.

We have built a generic library for assisting in capturing the state of the application, however the exact code to capture the state is dependent on the application.

5.4 Experience with MySQL

This section reports on the experience of applying PIKE to find concurrency bugs in MySQL.

5.4.1 State summary functions

As explained in Section 5.2, PIKE checks whether the application exhibits a linearizable behavior by comparing the internal state of different executions of the application. Achieving this goal, requires producing a high-level representation of the internal state which is, in practice, achieved by implementing application-specific state summary functions.

To write these summary functions, each of which captures a different part of the state, we analyzed five important state components of MySQL (described in Section 3.2) and classified them into two categories according to what type of data structures they represent.

Most of the analyzed data structures fall into the *set* category, since they are collections of elements where their order does not matter. The exception was the *binary log* structure which consists of an append-only *sequence* where the order in which the elements are added needs to be captured by the summary function.

Starting with the state components that describe sets, their summary function needs to be invoked in all places in the source code where elements are added, removed or modified to or from any of these data structures. Despite the complexity of the state, locating these turned out not to be too complicated since the source code of MySQL is reasonably well structured and there are functions that encapsulate these operations which are called from different points in the code.

At each of these points we invoke a generic summary function for sets, which is designed to provide an efficient update and comparison operation. This function maintains a cumulative hash value for the set (S) which is initialized to zero at the beginning of the execution. Then, upon adding or removing an element e, the summary function captures a hash of the deterministic parts of the element being added or removed (H_e). In this step it is important to remove sources of non-determinism like timestamps that would lead to state divergence. In addition, since some of the data structures annotated contain elements consisting of pointers, in these cases, instead of hashing the pointers, we hash the elements they point to.

Then, the value of H_e is either added or removed to the cumulative set value S. Both adding and removing is done by XORing the new value with the previous cumulative value, i.e.:

$$S_{new} = S_{old} \oplus H_e. \tag{5.1}$$

This leads to a compact representation of the state of the set that allows for a trivial comparison operator simply by comparing hashes. In addition, operations that modify elements are handled by treating them as a sequence of an add and a remove operation.

For the *binary log*, this representation does not work because it does not capture the order in which elements were added to the sequence. Therefore, in such cases, because the structure represents a sequence, we capture the order of the elements by hashing the concatenation of the previous cumulative value with the new element.

$$S_{new} = SHA1(S_{old}||e). \tag{5.2}$$

Finally, we also needed to extend this scheme to support containment instead of equality checks for sets. This can be easily achieved by replacing the cumulative XOR of the hash values with a counting Bloom filter [BMP+06]. Alternatively, we can just list all the elements in the set and compare them exhaustively, which is what is done by our implementation.

5.4.2 Input generation

Like other dynamic bug finding tools, our testing technique requires exploring different inputs in an attempt to find situations in which the application behaves incorrectly. Therefore we must find a diverse set of concurrent database operations that stand a good chance of triggering bugs. Again, the rich semantics and wide interface of MySQL make it particularly challenging given that we can only practically explore a small subset of all possible inputs.

We considered different options for generating test inputs. The obvious option is to generate the inputs manually; however this can be tedious and impractical for applications like MySQL. Another option is to randomly generate inputs, possibly with the aid of grammars that steer the input generation into generating inputs that are considered more useful. This option suffers from the problem that it is not straightforward to instrument the grammar in such a way that it creates multiple concurrent requests that are likely to cause contention for some particular part of the state of the application. A third option is to use tools that analyze the application, try to understand its behavior, and then attempt to automatically generate useful inputs [GKS05, CDE08]. However, while these tools work well for small and medium size applications, it is unclear if they can currently scale to the size of a codebase like MySQL.

Therefore we pursued a fourth option. MySQL already contains a large test suite, which has been manually created by the developers and testers of the application. Some of the tests were added specifically to prevent previous bugs from recurring in subsequent versions of the application. However, these tests are sequential tests and therefore would not be useful for finding concurrency bugs. Our solution was to convert these sequential tests into concurrent tests by breaking up the sequence of requests contained in a test and executing them concurrently by separate clients.

When deciding how many concurrent clients to use in our tests, we took into account that studies show that a significant amount of the concurrency bugs found only require a small number of threads to be triggered (typically two) [LPSZ08]. A separate study also showed that only a small number of requests is sufficient to expose bugs [SCA09]. Taking these factors into consideration, and to make the process more efficient, we generated tests involving two clients and with a limited number of requests per client (typically less than ten requests and starting from an empty database).

The original complete test suite contained approximately $50,000$ requests, as counted by the number of semicolons. Using our approach we manually converted around 5% of those requests into concurrency tests, thus generating 1550 pairs of inputs from concurrent threads.

Other approaches could be developed for generating inputs to test concurrent software. In Chapter 8, we discuss a future research direction to address the input generation aspect of concurrent testing.

5.5 Results

In this section we present the results of our experience of applying PIKE to MySQL.

5.5.1 Development effort

The first result we report on was the the amount of effort needed to understand the code of MySQL and develop the *state summary functions*. The annotations we inserted added up to 600 lines of code, as counted by the number of semicolons. This represents less than 0.2% of the number of semicolons in the MySQL source code.

While annotating the source code of MySQL, most of the effort was spent understanding the source code. We spent a total of about two man-months in the process of understanding both the structure and semantics of the application and annotating the source code.

5.5.2 Bugs found

We ran PIKE on MySQL opportunistically in a shared cluster using multiple machines (up to 15 machines). Each machine in the cluster had an AMD Opteron 2.6 GHz processor, 3 GB of RAM and was running a distribution of Linux with kernel version 2.6.32.12.

We tested MySQL by running it on 1550 inputs and for each input we configured PIKE to explore 400 different interleavings using its scheduler. The experiment lasted for about one month. Our implementation of PIKE could be optimized to reduce the computational cost in several ways. In particular, we could avoid going through the initialization phase of MySQL for each run by taking advantage of snapshotting techniques. Another way of speeding up testing could be to run PIKE on the target application previously compiled with optimization flags. The few inputs for which suspicious behavior is observed could then be re-executed with additional debugging support (on the version of the target application not optimized and with application-level debugging options enabled).

During our testing experiments PIKE was able to identify a total of 12 inputs that triggered concurrency bugs. Table 5.1 presents an overview of the inputs that we found to trigger incorrect behavior and in the following subsections we present our findings in more detail for different types of bugs, categorized according to their effects.

In some cases, we had different inputs that triggered bugs that showed similar effects. Because it was difficult for us to classify whether they correspond to the same bug or not, we decided to present the results in a more objective way by presenting in detail all of the inputs and effects of the bugs we found, instead of trying to count the number of distinct bugs. We then speculate about which of those inputs are likely to be triggering what could be considered the same bug.

Table 5.2 lists the various inputs that were flagged as positives by PIKE and that we confirmed to be caused by concurrency bugs. The table presents the requests that were concurrently executed in the test cases that triggered concurrency bugs together with the number of distinct thread schedules in which the program exhibited the incorrect behavior. Additionally, we also present information about the state and the output that were observed. Specifically, the table indicates whether the output of the concurrent execution matches the output of the sequential executions (O_A and O_B) and whether the state at the end of the concurrent execution matches the state at the end of either of the sequential executions (S_A and S_B).

Given the linearization assumption, PIKE flags a concurrent execution as having triggered a concurrency bug if it cannot find a sequential execution (X) that produces both a matching output and a matching final state (i.e., that has O_X=”Yes” and S_X=”Yes”). We can see that all entries in Table 5.2 fail to meet this condition.

In addition to discrepancies in the output or the state of the different interleavings, we also found some cases where the execution of the application blocked, which might have been caused by deadlocks, and cases where the application crashed. We have not analyzed these cases, but they are less interesting from our standpoint since these potential bugs would also have been found by other tools like CHESS [MQB+08], or tools that are designed to find deadlock bugs [NPSG09].

In our experiments, we did not come across non-concurrency bugs, and this is not surprising for two reasons. First, we used inputs that were based on the existing regression tests contained in the MySQL source code, and therefore MySQL should have been previously tested for these or very similar inputs. Second, a non-concurrency bug, if triggered would have likely produced the same wrong results in all interleavings, regardless of the interleaving being sequential or not, and therefore our detector would not have flagged it.

One point we would like to highlight about these results is that we used a testing suite that has been applied repeatedly, albeit in a way that runs inputs sequentially. We postulate that it might be possible to be even more effective if we use a different set of inputs. The downside is that, because we focused on what is not the latest version of MySQL, we found that some of the bugs have already been fixed, as we will detail next.

Next, we analyze in more detail the results for the two categories of bugs that our technique is aimed at: violations of the application semantics, and latent bugs. We

External effect	Non-latent	Latent	Total
Error	2	0	2
Semantic	2	8	10

Table 5.1: Number of inputs found to trigger concurrency bugs according to latency and external effects.

further divide the first category into semantic bugs and error bugs, depending on whether the violation of the intended semantics corresponds to an incorrect but non-error reply, or a more explicit error.

Requests	#	**Output**			**State**		
		O_A	O_B	**Effect**	S_A	S_B	**Latent**

CREATE TABLE t2 LIKE t1; ‡
INSERT INTO t2 SELECT * FROM t1; ‡
	9	✗	✗	Error	✗	✓	Non-latent

INSERT INTO t3 VALUES (1,'1'),(2,'2');
SELECT DISTINCT t3.b FROM t3,t2,t1 WHERE t3.a=t1.b; †
	1	✗	✓	Semantic	✗	✗	Latent

CREATE TABLE t2 LIKE t1; ‡
INSERT INTO t2 SELECT * FROM t1; ‡
	2	✗	✗	Error	✗	✓	Non-latent

TRUNCATE TABLE t1;
SELECT * FROM t2;
	35	✓	✗	Semantic	✗	✗	Latent

INSERT INTO t1 (a) VALUES (10),(11),(12);
SELECT a FROM t1;
	2	✗	✓	Semantic	✗	✗	Latent

INSERT INTO t2 VALUES (2,0);
SELECT STRAIGHT_JOIN* FROM t1, t2 FORCE (PRIMARY); †
	3	✓	✗	Semantic	✗	✗	Latent

DROP TABLE t1;
SHOW TABLE STATUS LIKE 't1';
	238	✗	✗	Semantic	✓	✓	Non-latent

INSERT INTO t1 VALUES (1,1,"00:06:15"); †
SELECT a,SEC_TO_TIME(SUM(t)) FROM t1 GROUP a,b; †
	1	✗	✓	Semantic	✗	✗	Latent

CREATE TABLE t2 SELECT * FROM t1;
DROP TABLE t2;
	17	✗	✗	Semantic	✗	✗	Non-latent

INSERT INTO t1 (a) VALUES (REPEAT('a', 20));
SELECT LENGTH(a) FROM t1;
	3	✗	✓	Semantic	✗	✗	Latent

INSERT INTO t1 VALUES (80,'pendant');
SELECT COUNT(*) FROM t1 WHERE LIKE '%NDAN%'; †
	2	✗	✓	Semantic	✗	✗	Latent

OPTIMIZE TABLE t1;
DROP TABLE t1;
	25	✓	✗	Semantic	✗	✓	Latent

Table 5.2: Properties of the triggered concurrency bugs that PIKE found. The table presents the number of concurrent executions that were flagged as positive for each of the inputs (#). Additionally it indicates whether the output of the concurrent executions matched the output of the sequential executions (O_A and O_B) and similarly for the state of the sequential executions (S_A and S_B). (Requests marked with † have been simplified for presentation purposes, the two identical pairs of requests marked with ‡ operate on distinct states)

Request 1	SHOW TABLE STATUS LIKE 't1';
Request 2	DROP TABLE t1;

Table 5.3: Requests responsible for triggering the sample semantic bug

Semantic bugs

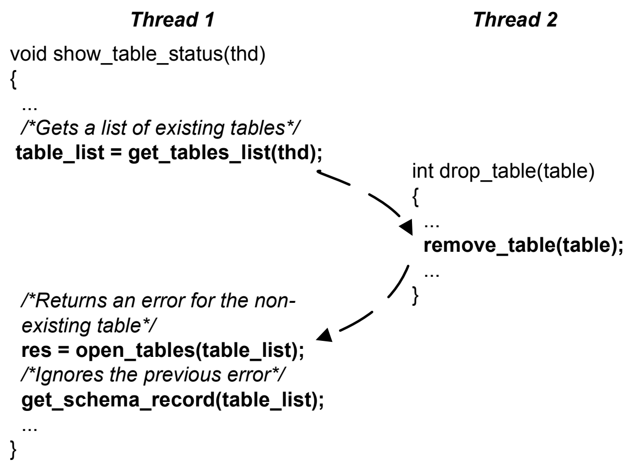
Figure 5.3 illustrates a representative example of a semantic concurrency bug in MySQL that was found by our detector. In the figure the arrow indicates the interleaving that triggers the bug. This bug is triggered when the server receives a specific *SHOW TABLE* request and a *DROP* request concurrently as shown in Table 5.3. Figure 5.3 shows a simplified snippet of the source code that is involved in this concurrency bug. The first thread, while executing the *SHOW TABLE* request obtains a list of names of tables. According to the semantics of the database this returned list should contain the names of all the tables in the database whose name contains the string "t1". But, if before the first thread processes the list of tables names the second thread is able to execute the *remove_table()* function, the open table list becomes obsolete. This in turn means that when the first thread resumes execution it will try to call the *open_tables()* function with an argument that contains obsolete data and will not be able to access the table that was dropped. The result is that the second thread will return to the user a success message for the *DROP* request. However, the first thread will return an entry, for the now non-existent table, indicating that it exists but some of the entries will contain the value NULL.

We note that this particular instance of a semantic bug was eventually reported in the MySQL bug report database, and patched in a version that succeeded the one we tested. However, it is important to note that we did not use that information during the process of generating inputs.

Other semantic bugs provided wrong results in even more subtle ways. For example, there were bugs where the application would simply provide wrong results based on stale data.

Error bugs

A sub-class of the semantic bugs that we found can be labeled as error bugs. We considered bugs to be error bugs if they manifest themselves by returning to the client an explicit error message that is not appropriate given the requests that were executed. During our experiments we found two cases in which error concurrency bugs were triggered.



```
void show_table_status(thd)
{
 ...
 /*Gets a list of existing tables*/
 table_list = get_tables_list(thd);

 /*Returns an error for the non-
 existing table*/
 res = open_tables(table_list);
 /*Ignores the previous error*/
 get_schema_record(table_list);
 ...
}
```

```
int drop_table(table)
{
 ...
 remove_table(table);
 ...
}
```

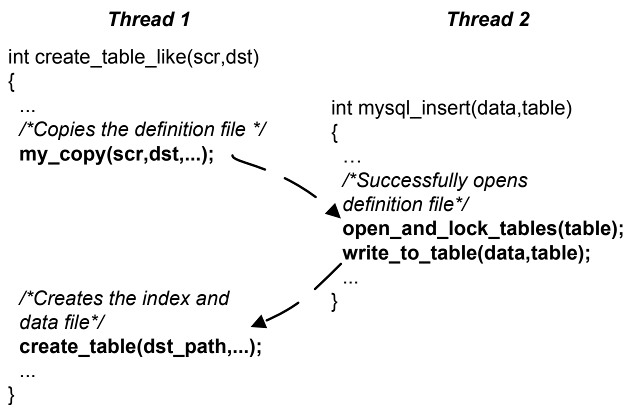
Figure 5.3: Sample semantic bug

Request 1	CREATE TABLE t2 LIKE t1;
Request 2	INSERT INTO t2 SELECT * FROM t1;

Table 5.4: Requests responsible for triggering the sample error bug

Table 5.4 presents the concurrent requests that were found to be responsible for one of the error bugs. This bug is triggered when one of the threads attempts to execute a *CREATE LIKE* request, which is supposed to create a new and empty table with a schema that is identical to another existing table, and a specific *INSERT* request that copies data from the existing table into the new table. As illustrated in Figure 5.4, the first thread, while handling the *CREATE* request, first copies the definition file containing the schema for the existing table. According to the synchronization logic in MySQL, the second thread is allowed to execute the *INSERT* request even before the first thread creates the index file and data file. Because of this, while executing the *INSERT*, the second thread is unable to open the data file and returns an error to the user stating that the *data file* does not exist instead of either succeeding (by writing data) or returning a different error stating that the *table* does not exist.

This example illustrates an important point – error bugs can also be subtle and difficult to distinguish from a correct execution, despite the fact that they return an error. One reason for this is that often an error message is a legitimate outcome of the operation, but the concurrent execution returns the *wrong* error message. Therefore, and unlike a situation where the application crashes or an assertion fails, we must know application-specific semantics to determine if an error reply is incorrect or not, and PIKE has proven to be effective in determining this.



```
int create_table_like(scr,dst)
{
  ...
  /*Copies the definition file */
  my_copy(scr,dst,...);

  /*Creates the index and
  data file*/
  create_table(dst_path,...);
  ...
}
```

```
int mysql_insert(data,table)
{
  ...
  /*Successfully opens
  definition file*/
  open_and_lock_tables(table);
  write_to_table(data,table);
  ...
}
```

Figure 5.4: Sample error bug

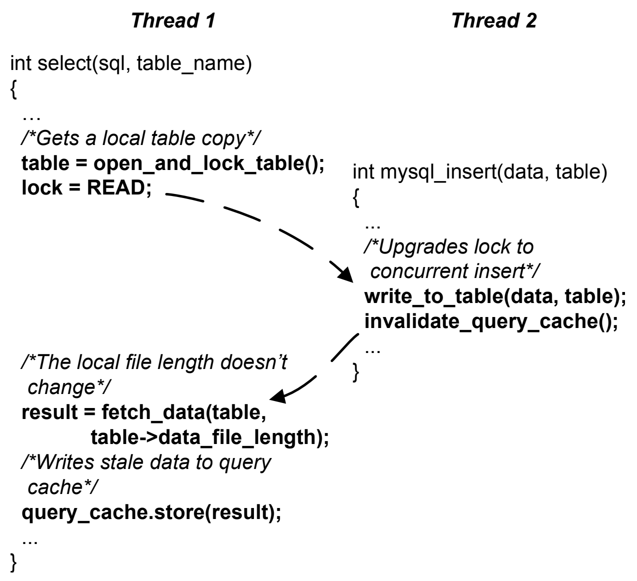
Latent bugs

Surprisingly, PIKE was able to find eight different situations that triggered latent concurrency bugs. All of the latent bugs we found had the external effect of providing wrong results in subtle ways and involved the *query cache* structure. As we will describe in Section 5.5.3, we also found situations where the *binary log* appeared to contain an incorrect state, but we were not confident that these represented bugs (i.e., that the incorrect state would lead to incorrect behavior visible by users) and so we did not flag them as such.

As an example, one of the cases where a latent concurrency bug is triggered occurs when the requests in Table 5.5 are executed concurrently. The simplified source code relevant to this example is shown in Figure 5.5. While executing the *SELECT* request, the first thread opens the table, locks it, and in the process makes a copy for itself of the state of the table. The logic of the application allows the second thread to then concurrently insert entries at the logical end of the table. However, when the first thread resumes execution it will rely on its local (and now stale) copy of the state of that table to fetch data. In the process, the first thread will skip the newly inserted entry and provide the old results to the client without immediately violating the semantics of the application (i.e., the returned value would be consistent with the first thread having executed before the second thread). In this bug, the actual semantic violation arises from the fact that the first thread also stores the stale data which the second thread does not invalidate in the *query cache*. This means that a third thread could, at a later point in time, read the stale data from the *query cache* and expose it to the clients, violating the expected semantics of the application.

Request 1	SELECT a FROM t1;
Request 2	INSERT INTO t1 (a) VALUES (10), (11), (12);

Table 5.5: Requests responsible for triggering the sample latent bug



Thread 1

```
int select(sql, table_name)
{
  ...
  /*Gets a local table copy*/
  table = open_and_lock_table();
  lock = READ;

  /*The local file length doesn't
   change*/
  result = fetch_data(table,
           table->data_file_length);
  /*Writes stale data to query
   cache*/
  query_cache.store(result);
  ...
}
```

Thread 2

```
int mysql_insert(data, table)
{
  ...
  /*Upgrades lock to
   concurrent insert*/
  write_to_table(data, table);
  invalidate_query_cache();
  ...
}
```

Figure 5.5: Sample latent bug

We saw the same pattern of latent bugs causing stale entries to be left in the query cache in other test cases, and we again stress that it is likely that some of the situations that triggered latent concurrency bugs could be triggering what could be considered the same bug. However, given the complexity of the application logic to both invalidate the *query cache* and to prevent certain specific concurrent requests from inserting simultaneously entries into the *query cache*, it is hard to state whether we are dealing with the same bugs objectively. Nevertheless it should be noted that the various cases that triggered latent bugs can be caused by very distinct types of requests, as can be seen in Table 5.2.

5.5.3 False positives

After the initial tests, approximately one third of the inputs generated potential false positives. Since this high fraction of false positives would make the analysis of the results impractical, we had to insert two filters to reduce the number of false positives which proved to be very effective. These filters allow testers to avoid false positives when the application deliberately violates linearizability.

The first filter we inserted was related to the table cache. Concurrent requests that try to open the same table concurrently will create distinct but identical entries in the table cache, whereas the same requests executing in sequence can reuse each other's entry. Therefore, we inserted a filter by checking that the table cache contents of the linearized execution is contained in the table cache contents of the concurrent execution.

The second filter was related to the query cache, and the fact that MySQL sometimes conservatively decides not to cache entries in the query cache when two concurrent requests are executed, one of them is a query, and the other would invalidate the entry for that query in the query cache. In this case, our filter ensures that a concurrent execution is flagged as negative (with respect to the query cache) if the query cache entries in the concurrent execution are contained in the set of entries in the linearization. Note that these may be considered performance bugs (or, at least, missed opportunities for a performance optimization), and this shows that PIKE might also be useful for analyzing and improving performance issues that may affect the application.

After inserting these two filters, the total number of false positives reported was 27. Of these, 22 are related to unexpected interactions between the framework and the application. In particular, some requests took a longer amount of time to complete which, in turn, caused an execution timeout in our framework to expire. In other cases false positives were caused by non-determinism in the reply that we had not caught (e.g., calls to the current time or random number generation). A third type of false positives was caused by timeouts in the NFS volume in which our results were written, which affected the output. All of these types of false positives were reasonably easy for us to diagnose.

The remaining five false positives involved a more careful analysis. These were caused by *binary log* entries being reordered (i.e., MySQL would change some internal structures in one order and the *binary log* in another order). This turned out to be acceptable under some circumstances. Typically this happened with pairs of concurrent requests in which one of the requests executed an optimization or maintenance task (e.g., *OPTIMIZE* and *FLUSH* requests). The fact that these operations affect the performance but not the results implies that, when the *binary log* state is required (normally when a replica recovers from a fault), repeating these entries in the wrong order will not affect the output of the operations, but only the moment in the sequence of re-execution of these operations when the performance optimizations are performed.

5.6 Limitations and discussion

This section discusses the consequences of relying on modifications to the tested software and the potential of PIKE to extend existing test-suites.

5.6.1 Reliance on modifications to tested software

As mentioned in Section 5.2, the current design of PIKE requires changes to the tested applications, namely to implement the state summary functions. Unfortunately, this requirement constitutes a disadvantage, for two main reasons, that we discuss next.

First, software modifications require work from developers, leading to an increase in the adoption cost which, in turn, dissuades the adoption of the testing tool. However, our experience with PIKE showed that, given the specific changes required by our approach, the software modifications required to test MySQL involved only a moderate effort (Section 5.5.1). In addition to the limited effort, because PIKE allows the modifications to be performed incrementally on a per-data structure basis, our proposed testing approach mitigates this problem by allowing developers to control the tradeoff between the scope of testing (i.e., amount of state analyzed) and effort required for testing.

Second, modifications to the tested software prevent developers from achieving their real goal, which is to test the original software. In some cases this difference can be problematic because software modifications can potentially introduce bugs or simply affect the application behavior in a manner that masks existing bugs. However, in the case of PIKE, because the major modifications required (the state summary functions) are expected to run only at the end of the execution[1], after the output has been sent to users, it is unlikely that such changes would negatively affect the observed output; even though they could still crash the application or produce wrong information about the state of the tested application.

Despite the limited impact of the specific modifications required by PIKE, our real-world experience testing software allowed us to better understand the importance of avoiding this requirement. In this context, Chapter 6 proposes a testing tool, SKI, that entirely forgoes the requirement to modify the tested software by means of a dedicated design. Section 7.1.2 takes a step back to discuss the general problem of modifying tested software in the broader context of software testing. In addition, Section 8.1 discusses a

[1]The small modifications required by PIKE to identify three location in the source code (Section 5.3.2), for the purpose of controlling the interleavings, represent negligible modifications given their size and simplify.

possible research direction to improve the design of PIKE with respect to the need to modify the application.

5.6.2 Extending traditional test suites

PIKE could be leveraged to extend traditional testing approaches, which generally focus on non-concurrency bugs. Existing test suites could adopt PIKE to find concurrency bugs, in addition to non-concurrency bugs, by including both PIKE's component to detect the concurrency bugs, as well as a custom scheduler to explore the interleaving space. Importantly, PIKE would preclude developers from having to specify the correct output produced by each of the concurrent tests, which is a burdensome, albeit common, practice in the case of traditional non-concurrent tests. Furthermore, because PIKE analyzes the application internal state, in addition to the output, it is able to detect latent concurrency bugs.

5.7 Summary

This chapter presented PIKE, a tool for testing concurrent applications that finds two particularly challenging types of concurrency bugs: semantic concurrency bugs and latent concurrency bugs. We applied PIKE to a mature version of MySQL and, in the process, we were able to find several semantic and latent concurrency bugs. In addition, we found that it was simple to write the necessary annotations to capture an abstract view of the service state, and that it was easy to make the number of false positives tractable by writing simple filtering rules for common violations of linearizability at the level of the application state. This work allowed us to conclude that it is feasible to extract a specification from a complex applications, such as MySQL, by presuming that, in the general case, developers intend to provide linearizable semantics.

6 Exposing concurrency bugs in kernels

6.1 Overview

In the current multi-core era, kernel developers are under permanent pressure to continually increase the performance of kernels through concurrency. Examples of such efforts include reducing the granularity of locking [Rus], rewriting subsystems to use parallel algorithms [CKZ13a], and using non-traditional and optimistic synchronization primitives (such as RCU [MS98] and lock-free data structures [Val94]). Unfortunately, previous experience has shown that all these efforts are error-prone and can easily lead to kernel concurrency bugs — bugs that are only exposed by a subset of the possible thread interleavings.

In practice, kernel developers find concurrency bugs mostly through manual code inspection [WJKT05, Hol14] and stress testing [BAEFU06a, Sto02] (i.e., applying intense workloads to increase the chances of triggering concurrency bugs). While useful, both approaches have significant shortcomings: code inspection is labor-intensive and requires significant skill and experience, and stress testing despite having low overhead and being amenable to automation, offers no guarantees and can easily fail to uncover difficult to find concurrency bugs — i.e., edge cases that are only triggered by a tiny subset of the interleavings. It thus stands to reason that kernel developers could benefit from tools without these limitations.

To this end, we propose a *complementary* testing approach for automatically finding kernel concurrency bugs. Our approach explores the kernel interleaving space in a systematic way by taking full control over the kernel thread interleavings. This approach has been explored for user-mode applications, namely by PIKE (Chapter 5), and existing literature has proven that it can yield good results [MQB$^+$08, BKMN10, NBMM12a]. But, unfortunately, the systematic approach has not yet been applied to commodity kernels because achieving control over the thread interleavings of kernels involves several challenges. First, to be practical, a concurrency testing tool must be generally applicable, rather than being specific to a particular kernel or kernel version, which precludes kernel-specific modifications. Second, the kernel is the software layer that implements its own thread scheduler, as well as the thread abstraction itself, making the external control of thread interleavings non-trivial. Finally, to be effective, such a tool must be able to control kernel interleavings while introducing a low overhead.

In this chapter, we report on the design and an evaluation of SKI[1], the first tool for the systematic exploration of kernel interleavings to overcome these challenges. To achieve control over kernel interleavings in a portable way, SKI uses an adapted virtual machine monitor that (1) determines the status of the various threads of execution, in terms of being blocked or ready to run, to understand the scheduling restrictions, and (2) selectively blocks a subset of these threads in order to enforce the desired schedule. Notably, these key tasks are achieved without any modification to the kernel and without specific knowledge of the semantics of the kernel's internal synchronization primitives. Furthermore, we propose several optimizations, both at the algorithmic and at the implementation levels, that we found to be important for scaling SKI to real-world concurrency bugs.

We evaluated SKI by testing several file systems in recent versions of the Linux kernel and we found 11 previously unknown concurrency bugs. Of these, several concurrency bugs can cause serious data loss in important file systems (*ext4* and *btrfs*). We also show how SKI can be used to reproduce concurrency bugs that have been previously reported in two different operating systems (Linux and FreeBSD), and compare SKI's performance against the traditional stress testing approach.

We believe that SKI is an important step towards increased kernel reliability on multicore platforms. Nonetheless, there remains significant room for exploiting domain- and kernel-specific knowledge. For instance, in this dissertation we propose a scheduling algorithm, which generalizes PCT (Section 3.1.1), that is generic in the sense that it makes no assumptions about the kernel under test. However, based on the SKI infrastructure, other kernel-specific scheduling algorithms could be implemented, for example, to restrict the interleavings explored to those that affect specific kernel instructions, such as code that was recently modified. Thus, we believe that SKI can provide benefits even beyond those described in this dissertation, since it can serve as an experimentation framework for different systematic techniques.

The rest of the chapter is organized as follows. Section 6.2 motivates the need for better kernel testing tools. Section 6.3 presents the design of SKI. Section 6.4 proposes several optimizations to make SKI scale to real-world concurrency bugs. Section 6.5 describes the details of our implementation and Section 6.6 evaluates SKI on commodity kernels.

[1]Systematic Kernel Interleaving explorer

Section 6.7 discusses some of its limitations and Section 6.8 presents a summary of the chapter.

6.2 Kernel API and kernel modifications

Knowing which threads are live and being able to exercise control over the interleavings are two essential mechanisms to systematically explore the interleaving space (Section 3.1). In the case of user-mode applications, both of these essential mechanisms can be easily and portably implemented in a proxy layer (e.g., through *LD_PRELOAD* or *ptrace*) by intercepting all relevant synchronization primitives to infer and override the liveness state of each thread [MQB+08, BKMN10, NBMM12a]. For example, PIKE uses *LD_PRELOAD* to achieve this level of control over applications and to implement the PCT scheduling algorithm. Unfortunately, this approach does not work for kernels because kernels do not have a portable interface to allow such fine-level control over its threads by a testing application. An alternative would be to include the testing tool within the tested kernel by modifying the kernel itself, but this approach would suffer from several disadvantages:

- **Lack of portability and API instability.** Any dependency on kernel-internal APIs would a priori limit the portability of the envisioned testing tool, preventing its seamless application across different kernels and even across different versions of the same kernel. In contrast to well-documented, standardized user-space interfaces (e.g., the pthreads API), the internal API of most kernels is not guaranteed to be stable, and in fact typically changes from version to version. In particular, given the current trend towards increased hardware parallelism, kernel synchronization has generally been an active area of development in Linux and other kernels [MS98, CKZ13a].

- **Complexity of the internal interface.** An additional problem with the internal API of the kernel, also noted in previous work [EMBO10], is that the semantics of in-kernel synchronization operations are particularly complex. Furthermore, the exact semantics of such operations tend to differ from kernel to kernel. This calls for solutions that do not require a detailed understanding of these semantics.

- **Other forms of concurrency.** Interrupts are pervasive and critical to kernel code. However, exercising fine-level control over their timing from within the kernel itself would be particularly challenging, as interrupts are scheduled by the hardware.[2]

- **Intrusive testing.** Requiring modifications to the tested software goes against the principles of testing [KL93] — testing *modified* versions of the software can potentially introduce or elide bugs.

In the next section we explain how SKI overcomes these challenges while enabling the systematic exploration of kernel thread interleavings.

6.3 SKI: Exploring kernel interleavings

This section presents the design of SKI. We start by providing an overview of our solution (Section 6.3.1), and then we describe how SKI exercises control over thread interleavings (Section 6.3.2) and how it gathers the necessary liveness information (Section 6.3.3). We conclude this section with a description of the scheduling algorithms employed, i.e., the interleavings chosen for each run (Section 6.3.4).

6.3.1 Overview

The inputs given to SKI are the initial state of the system under test and the kernel input that is to be tested concurrently (i.e., two or more concurrent system calls). Given these inputs, SKI carries out several test runs corresponding to different concurrent executions, where each test run is fully serialized, i.e., the tool enables only a single thread to execute at each instant. This enables precise control over which interleavings are executed, and allows SKI's scheduler to choose successive runs to improve the interleaving space coverage. Either during or after each test run, a bug detector is used to determine if the test has flagged a possible bug. Such bug detectors can perform simple, generic actions like detecting crashes, or complex, application-specific actions like running a system integrity check after the test run.

As mentioned in the previous section, for SKI's scheduler to gain control over the interleavings executed by the kernel, it must perform two key tasks: inferring thread liveness and overriding the scheduler. To accomplish both without modifying the OS kernel under test, we implement the scheduler of SKI at the level of a modified virtual machine monitor (VMM), taking as input a virtual machine (VM) image that incorporates the initial state of the kernel immediately before the system calls are invoked

[2]While user-mode signals are similar to interrupts, many programs do not use signals and therefore existing user-mode tools, including PIKE, do not handle them [MQB+08, BKMN10].

concurrently. Implementing the scheduler at the VMM level enables it to both observe and control the kernel under test.

This advantage comes, however, at the cost of making it more difficult to implement the two aforementioned key tasks. This is because, at the VMM level, the hypervisor observes a stream of machine instructions to be executed, and has direct access only to the physical resources of the underlying hardware (such as registers or memory contents). These low-level concepts are distant from the abstractions that are implemented by the kernel in software, such as threads and their respective contexts. Furthermore, it would intuitively seem necessary to have access to these abstractions for suspending the execution of a thread and replacing it with another thread.

6.3.2 Exercising control over threads

To control the progress of threads, SKI relies on the observation that many important kernels (e.g., Linux, Windows, MacOS X, FreeBSD) include a mechanism to allow applications to *pin* threads to individual CPUs (i.e., to specify the thread affinity), as described in Section 3.3. This mechanism, provided by kernels to user-mode applications for performance reasons, can be exploited to create a 1:1 mapping between threads (a kernel abstraction) and virtual CPUs (an ISA component, controllable by the VMM). This mapping in turn allows SKI to block and resume a thread execution by simply suspending and resuming the corresponding virtual CPU's execution of machine instructions.

To implement the mapping between threads and CPUs, SKI includes, in addition to the modified VMM, a user-mode component that runs inside the VM and issues system calls to pin threads to virtual CPUs (see Section 6.5.3). Note that for scalability reasons, each test generally involves only few threads, and hence it suffices to configure a small number of virtual CPUs. Section 6.7.1 discusses the limitations of this mapping approach regarding a particular category of threads, the kernel threads, and the implications on testing the scheduler code itself.

6.3.3 Inferring liveness

To explore the interleaving space, SKI requires information about whether threads are blocked or able to progress, analogously to what is required by the existing user-mode tools [BKMN10, MQB+08, NBMM12a]. This requires SKI to be able to identify constructs such as spin-locks or barriers, where a CPU executes a tight loop, constantly checking the value of a memory location for changes. SKI would be impractical if it were not able to detect such constructs, for several reasons. First, executions would take longer because more instructions would be executed (e.g., iterations of a spin loop). Second, because more instructions would be executed, the space of possible interleavings would significantly increase, since the number of possible interleavings is exponential in the length of the test. Third, and most importantly, given the scheduling algorithm that we describe in Section 6.3.4, two interleavings could be considered different even when they only differ in the number of iterations executed by the polling loop of a spin lock. This would be detrimental to the efficiency of SKI, since many of the explored schedules would be effectively equivalent.

The difficulty in inferring thread liveness is that, from the point of view of the VMM, CPUs are constantly executing instructions. As such, it is difficult to distinguish the normal execution of a program from a polling loop.

One possible solution that we considered, but ultimately rejected, relies on annotating the kernel by specifying the locations within the kernel code where the CPU executes instructions without making any actual progress, namely situations where the kernel is waiting for some event external to the CPU (such as an action performed by some other CPU or a device notification). However, this approach would be laborious, error prone, and non-portable.

Instead, we found several simple heuristics independent of the kernel code that enable the VMM to infer whether a CPU is making progress or not.

H1: Halt heuristic. The first heuristic flags the CPU as non-live when it executes the halt instruction (HLT).[3] According to the instruction set specification, HLT marks the CPU as waiting for interrupts. This instruction is typically used by kernels to implement, in an energy efficient way, the idle thread when the kernel scheduler has no other threads to run. When the CPU subsequently receives an interrupt, it is marked as live again.

H2: Pause heuristic. The second heuristic relies on the observation that kernels use the pause instruction (PAUSE) to efficiently implement spin-locks. In the x86 architecture, the pause instruction has been introduced to avoid wasting bandwidth on the memory bus when a CPU goes into a tight polling loop, and therefore its execution is a good indication that the CPU is spinning on a lock. Thus, when our modified VMM detects the execution of two nearby pause instructions, i.e., within an instruction window of size h_2, it considers the CPU to be non-live and takes note of the memory read-set associated with the instructions executed between the two pause instructions.

[3]We focus on the ubiquitous x86 architecture in this dissertation; the presented ideas, however, can be similarly applied to other architectures.

Pause instructions in close proximity are detected by the VMM by checking, at every pause instruction, whether another pause instruction was recently executed. Later on, when another CPU changes one of the addresses in the read-set, the non-live CPU is optimistically marked as live again.

H3: Loop heuristic. The third heuristic detects situations where the CPU is waiting for some external event, but that are not caught by the second heuristic. This could happen if, for example, a spin-lock were implemented without including the pause instruction. To detect CPUs stuck in a polling loop, our modified VMM maintains a window, of size h_3, of the last few instructions executed by each CPU. If a CPU repeatedly executes the same instructions (i.e., if it executes a loop), and if an instruction in the loop repeatedly reads the same value from the same memory address, the executing context is flagged as non-live after a certain number of loop iterations. Again, SKI takes note of the read-set of detected polling loops to later re-enable the CPU.

H4: Starvation heuristic. As a last resort, in case the above heuristics are not able to detect situations where there is no progress, SKI keeps a count of the number of instructions executed continuously by the current CPU, and, if it exceeds a threshold (h_4), it conservatively presumes that the CPU is no longer making progress. The CPU is marked live again after a certain number of instructions have been executed by the other CPUs. This heuristic ensures the detection, for example, of loops that are missed by H3 if h_3 is set smaller than the loop size.

We determined the values for the thresholds of these heuristics, which remained constant throughout all our tests, through simple experimentation. From our experience, these mechanisms were sufficient to ensure the effectiveness of SKI for a wide range of kernel versions, at both reproducing previously known bugs and at finding unknown bugs.

6.3.4 Scheduling algorithm

SKI executes a VM multiple times under different schedules to ensure interleaving diversity across the runs. To select and prioritize the interleavings that are to be explored SKI needs to implement a scheduling algorithm. To this end, we have generalized the PCT algorithm (Section 3.1.1) to support testing operating system kernels. SKI generalizes PCT by supporting interrupts (Section 6.3.4), which is a fundamental requirement for testing kernels.

We consider the proposed algorithm to be just one instance from a range of possible algorithms (albeit one that in our experience happens to work well), and developers that make use of the tool might consider adding other, more refined algorithms. For example, it may be possible to develop effective scheduling algorithms that exploit specific characteristics of kernel code. This section presents our generalization of the algorithm, and assumes that the reader is familiar with the original PCT algorithm, which is described in detail in Section 3.1.1.

The scheduling algorithm we propose differs from the PCT algorithm in two aspects. First, given that SKI operates at the level of the virtual machine monitor, our algorithm does not have access to the thread abstraction. Thus, instead of scheduling threads, our algorithm schedules CPUs. Second, given the interrupt mechanism within the kernel context, our scheduler has to decide when to dispatch interrupts.

Interrupts do not appear in the context of user-mode programs, but we need to control their schedule when testing the kernel for two different reasons. First, concurrency bugs may depend on the interleaving of interrupts, so our algorithm should be able to explore this part of the interleaving space. Second, interrupts are in some cases required for the successful completion of system calls, and therefore interrupts need to be scheduled to conclude the execution of the tests. For example, some system calls are only able to finish if, during their execution, other CPUs handle the TLB flush interrupt. Thus, since the scheduler of SKI must handle both threads and interrupts, we will refer to *contexts*, instead of threads, throughout the description of our generalized scheduling algorithm.

As the scheduler needs to consider when interrupts are handled, each CPU is tracked as being in one of two different contexts: it may either execute in the context of an interrupt handler (*interrupt-context*), or it may execute outside of the context of any interrupt handlers (*CPU-context*). Each *interrupt-context* is defined by the CPU on which it arrived and by the interrupt number that it represents. From the point of view of the scheduler, *interrupt-contexts* are created, and therefore become schedulable, when the corresponding interrupt arrives on its specific CPU. These execution contexts are, to our scheduler, the equivalent to threads for other systematic exploration algorithms, and as such they need to be detected by the scheduling logic. SKI infers the context by tracking the interrupt handler dispatches and the *IRET* instruction invocations (which are used to return from interrupt handlers). Figure 6.1 shows examples of schedules involving two *CPU-contexts* and one *interrupt-context*.

To achieve further control over the tests, SKI allows the user to specify a set of execution contexts that are allowed to run during the test. In particular, placing restrictions on the set of eligible execution contexts may be useful in specific testing scenarios, to restrict the scheduling space that is explored.

Schedule 1			Schedule 2		
CPU1	INT1	CPU2	CPU1	INT1	CPU2
INST 1			INST 1		
INST 2			INST 2		
INST 3			<res>		
INST 4				INST 1	
<end>				INST 2	
	INST 1			<end>	
	INST 2				INST 1
	<end>				INST 2
		INST 1			INST 3
		INST 2			<end>
		INST 3	INST 3		
		<end>	INST 4		
			<end>		

Figure 6.1: Two examples illustrating schedules produced by SKI. Each schedule involves three contexts, two CPU-contexts and one interrupt-context. Both schedules start with the same initial context priorities. However, Schedule 2 differs from Schedule 1 because it contains one reschedule (`<res>`).

6.3.5 Discussion

The design of SKI ensures correctness, meaning that SKI never causes the kernel to exhibit a behavior that could not possibly occur during normal executions of the kernel, because SKI exercises control over the kernel schedule by temporarily suspending the execution of instructions on chosen CPUs. Correct kernels have to be able to handle this mechanism because the hardware specification does not provide guarantees about the speed of the CPUs. Furthermore, modern kernels are expected to work well within virtual machines, where the apparent speed of CPUs is likely to not be regular simply because the host system might be under heavy load.

Despite this correctness guarantee, some bug detectors may still produce false positives (e.g., data race detectors). In such cases, regardless of how the interleaving space is explored, the obtained results require further analysis specific to the the employed bug detector.

6.4 Scaling to real-world kernels

The total number of possible schedules grows exponentially with the length of the code under test. For most programs, including the kernel, it is not practical to *exhaustively* explore all interleavings, and therefore it is important for concurrency testing tools to include mechanisms for increased scalability.

The p parameter, used by the scheduling algorithm (Section 6.3.4), constrains the schedules that may be explored and therefore improves scalability by bounding the number of possible schedules. This is done without much impact on the effectiveness of the testing tool, given the observation that, in practice, most bugs can be triggered with few reschedule points [BKMN10]. Similarly, it has been shown that many concurrency bugs can be triggered with a small number of threads [LPSZ08] and with a small number of concurrent requests [SCA09] – which are observations also leveraged by PIKE. Based on these observations, we configured SKI in our tests to use small values for these three dimensions (reschedule points, number of CPUs, and number of system calls).

Despite these optimizations, we noticed in our initial tests that SKI's scalability was limited by the fact that even a single system call can execute a large number of instructions — a single system call typically executes many thousands or even millions of instructions. This implied that, even if we limited SKI to $p = 1$, the number of runs that would be required to explore all schedules would be on the same order of magnitude as the number of instructions, which is impractically large.

To address this scalability issue, SKI relies on a technique first proposed by Godefroid [God97] that exploits the fact that some schedules are equivalent and thus redundant, as illustrated in Figure 6.2. In particular, we rely on the observations that (1) schedules that do not differ in terms of the relative order of communication points (where threads see the effects of each other) are observationally equivalent from the standpoint of the interleaved threads, and that (2) most of the kernel instructions do not constitute communication points between CPUs. Taken together, these two observations allow us to significantly improve SKI's scalability by restricting reschedules to occur only at communication points.

More precisely, we define a *point of communication* as an instruction that accesses a memory location that is also accessed by another CPU during the test, and where at least one of the accesses is a write. Such concurrent memory accesses can influence the final outcome of the execution: in the case of two concurrent writes, the last value to be written prevails, and in the case of a write concurrent with a read, the value read may or may not reflect the write, depending on the schedule. Other tools have also tried to

Schedule 1		Schedule 2		Schedule 3	
CPU1	CPU2	CPU1	CPU2	CPU1	CPU2
`A=1`		`A=1`		`A=1`	
`<res>`		`B=1`		`B=1`	
	`A=0`	`<res>`		`C=B+A`	
	`D=A+1`		`A=0`	`<res>`	
`B=1`			`D=A+1`		`A=0`
`C=B+A`		`C=B+A`			`D=A+1`
`PRINT C`		`PRINT C`		`PRINT C`	

Figure 6.2: Example showing two equivalent schedules (Schedules 1 and 2) and one schedule that is not equivalent to either of the others (Schedule 3). In this example, only variable `A` is used for communication between CPUs. Because variable `B` is accessed by only one CPU, placing the reschedule point (`<res>`) immediately before (Schedule 1) or immediately after (Schedule 2) the statement `B=1` does not change the result of the execution.

avoid equivalent schedules but by relying instead on identifying and preempting threads at locations involved in possible data races or at locations invoking synchronization primitives [MQB+08].

SKI gathers the location of possible communication points by monitoring memory accesses during the tests. During each run, it tracks the locations of the memory accesses, the CPU responsible for the accesses, and the types of accesses (read or write). After each run, SKI generates a set of program addresses that are potential communication points, and merges this information with an accumulated set of potential communication points for that specific test case. Note that this process does not rely on sample runs — every run monitors the memory accesses and, therefore, potentially learns new communication points. As this accumulated set is constructed, it is used in subsequent runs for the same test case to decide which schedules are equivalent, thereby limiting the set of instructions that qualify as reschedule points. During the initial run, because no communication points are known, SKI learns new communication points for subsequent runs without executing reschedule points.

In our experiments, we observed that, as expected, both data and synchronization accesses were identified as communication points. To give some examples, data accesses occur when both CPUs try to modify the same field in a shared structure (e.g., a file reference count), and synchronization accesses occur when both CPUs try to acquire the same lock. An advantage of SKI's dynamic approach is that whether or not an instruction qualifies as a reschedule point depends on the code that *both* CPUs actually execute (e.g., the specific system calls or interrupt handlers that are invoked). As a result, if two CPUs acquire different locks unrelated to the tested functionality, such accesses will not be considered communication points (in the context of the current test case).

In practice, SKI estimates the expected number of instructions, k (recall Section 3.1.1), based on previous runs. With the communication points optimization, instead of considering individual instructions when placing reschedule points, we consider only communicating instructions, and thus let the algorithm take coarser-grained steps in its exploration of the interleaving space. That is, by limiting the set of reschedule point candidates, the magnitude of the parameter k is effectively reduced. During the initial run, because no communication points are known, the initial value of k is ignored. In addition to these algorithmic optimizations, SKI includes several optimizations, at the level of the implementation, to ensure its effectiveness (Section 6.5.4).

6.5 Implementation

We implemented SKI by modifying QEMU, a mature and open-source VMM, and its JIT compiler. In total, our implementation added 13, 542 lines of source code to QEMU. We also built a user-mode testing framework consisting of 674 lines of source code to help users write test cases for SKI (Section 6.5.3). In addition, we implemented various scripts to set up and automate tests and also to analyze the gathered information.

6.5.1 Overview

SKI provides a helper tool to allow kernel developers to specify the concurrent system calls, by building a VM containing the corresponding test case (Section 6.5.3). When executed under SKI, this VM first goes through an initialization phase, performing test-specific actions to configure the system, and then signals the beginning of the test to the VMM using hypercalls (i.e., calls between the VM and the VMM). When all virtual CPUs have received the signal, the SKI scheduler is activated.

SKI's first action is to take a snapshot of the VM. The VM snapshot includes the entire machine state (memory state, disk state, CPU state, etc.) and thus allows SKI to run multiple executions from an identical initial state.

Starting from this VM snapshot, SKI places reschedule points and assigns starting priorities as described in Sections 6.3.4 and 6.4, and then resumes the execution of the highest-priority context and enforces the chosen schedule, thereby exploring different schedules on each run.

To mark the end of the test, the user-mode component inside the VM issues a hypercall to the VMM. Afterwards, the VM is allowed to run normally (i.e., without schedule restrictions) until the testing application asks to terminate the execution. This last phase is useful to let the user-mode component execute test-specific diagnostics (such as a file system check) inside the VM.

6.5.2 Runnable contexts

The scheduler of SKI allows, at any point in time, only the *live* and *active* context with the highest priority to run. The liveness of a context is inferred by the VMM according to the heuristics explained in Section 6.3.3; the criteria for determining whether a context is active or not depends on the type of context. A CPU-context is considered active if it has not reached the end of the test, which is flagged by the user-mode component using a hypercall, as discussed above, whereas an interrupt-context is considered active only after it has been triggered by the respective hardware device and before the corresponding IRET instruction has been executed.

6.5.3 Helper testing framework

We built a user-mode helper framework that allows users to easily build a testing VM ready to be used by SKI. It includes a user-mode application that runs inside the testing VM for the purpose of setting up the kernel and for providing the required test input (e.g., system calls).

The user-mode test framework automatically creates the testing threads/processes, pins each thread/process to a dedicated virtual CPU, issues the hypercalls to mark the beginning of the test (right before the test function is called) and the ending of the test (right after the test function returns), and finally requests the termination of the VM (when all post-test functions have completed). This framework can be used both to manually create test cases (Section 6.6.2) or to adapt existing test suites to leverage SKI for the interleaving exploration (Section 6.6.3).

We first implemented the framework targeting Linux and subsequently ported it to FreeBSD, and have been using it to conduct tests on both operating systems. The helper framework itself was easily ported because only few of the system/library calls it relies are not part of the POSIX standard (namely the calls to pin threads/processes, which have slightly different interfaces).

6.5.4 Optimizations and parallelization

In addition to the algorithmic optimizations described in Section 6.4, we have implemented several other optimizations to improve the performance of SKI. One of our main optimizations avoids resuming from a snapshot for each tested execution, which takes a few seconds in the original version of QEMU. Instead we have implemented in SKI a multi-threaded forking mechanism to take advantage of the copy-on-write semantics offered by the host OS, amortizing the cost of resuming from a snapshot over multiple executions. This mechanism has been implemented, without modifying the host kernel, by issuing a fork system call from a specific VMM thread (which only forks the calling thread) and subsequently manually spawning the other threads as well as reconstructing their respective stacks. To ensure that the fork system call is actually efficient, despite the large address space of SKI, our implementation additionally leverages large memory pages (2MB in x86-64 systems) at the host level.

The benefit of forking executions is not limited to executions that test the same input because we allow the testing application to receive, through a hypercall, a parameter that specifies the testing input. Thus, from a single snapshot, SKI can explore different inputs and different interleavings, making the overall cost of creating and resuming from a snapshot negligible.

In addition, given that in our testing scenario after each execution we discard most of the state of the VM (e.g., VM RAM and disk contents), we optimized SKI by converting several file system operations, performed by the original QEMU on the host, into memory operations.

Given that our workload is parallelizable, SKI takes advantage of multicore host machines by spawning multiple VMs to perform multiple concurrent tests. We have also implemented a testing infrastructure to distribute the workload across multiple machines, further increasing the testing throughput.

6.5.5 Bug detectors

Section 6.3 presented the algorithms and mechanisms that SKI employs to explore the thread interleaving space of the kernel. However, to find concurrency bugs an orthogonal problem needs to be addressed — it is necessary to identify which of these executions triggered bugs.

In Section 6.6 we show how SKI can be combined with different types of bug detectors — we evaluate SKI using bug detectors to detect crashes, assertion violations, data races and file system inconsistencies. Our implementation detects crashes and assertion violations by monitoring the console output at the VMM level. The detection of data races is also performed at the VMM level by recording racing memory accesses, similarly to DataCollider [EMBO10]. File system inconsistencies, in contrast, are detected by running existing file system checkers inside the VM itself after each test.

6.5.6 Traces and bug diagnosis

To enable the implementation of external bug detectors and to allow the diagnosis of bugs through manual inspection, SKI is able to produce detailed logs of the executions. These traces contain the exact ordering of instructions and the identity of the context responsible for the instructions. In addition, SKI can be configured to produce traces with all the memory accesses and the values of the main CPU registers.

We built some analysis tools that parse these traces to provide useful information. One of our tools extracts source code information (assuming the kernel is compiled with debugging symbols) and disassembles the kernel binary to annotate the trace with both source code and assembly instructions. We also implemented another diagnosis tool that generates the call graph for each execution. While none of these tools is conceptually particularly challenging, in our experience, they complement each other well and make the rich information collected by SKI much more accessible.

Apart from the traces produced by SKI, the bug detectors we built are another important source of diagnostic information. For example, the data race detector that we implemented identifies the exact memory address as well as the instruction addresses involved. As another example, the crash reports produced by the Linux kernel include a detailed stack trace that is very convenient for developers to diagnose bugs.

6.6 Results

This section evaluates the effectiveness of SKI in exposing real-world kernel concurrency bugs. After describing the configuration that we employed in our experiments, we report on our experiments using SKI to reproduce previously known bugs and comparing it with traditional approaches (Section 6.6.2). We then report our experience in applying SKI to recent and stable versions of the Linux kernel, which resulted in the discovery of several previously unknown concurrency bugs (Section 6.6.3).

6.6.1 Configuration

We conducted our experiments on host machines with dual Intel Xeon X5650 processors, each with 12 hardware threads, and 48 GB of RAM running Linux 3.2.48.1 as the host kernel. To increase the testing throughput, we configured SKI to run 22 testing executions in parallel on each machine and we ran our experiments on up to 12 machines at a time.

For each test case reported in this chapter, we configured SKI to use $p = 2$ and we explored 200 schedules in the large-scale experiments to find new bugs (Section 6.6.3) and $50,000$ schedules in the experiments to reproduce known bugs (Section 6.6.2). SKI's liveness heuristics used $h_2 = 30$, $h_3 = 20$ and $h_4 = 500,000$ (Section 6.3.3). We tested several different versions of Linux, ranging from 2.6.28 to 3.13.5, depending on the experiment, and one of the experiments described tested FreeBSD, version 8.0. Importantly, the same configuration of SKI was used in all tests: we did not have to modify any settings to adjust SKI to a particular tested kernel version, and we also did not have to modify the kernels under test.

6.6.2 Reproducing concurrency bugs

We evaluated the effectiveness of SKI in reproducing previously reported kernel concurrency bugs. To find typical bug reports, we searched the kernel Bugzilla databases, the kernel development histories (i.e., the *git changelogs*), and the mailing list archives. From these sources, we selected four independently confirmed kernel concurrency bugs. We opted for a diverse set of bugs that were particularly well documented. Furthermore, to enable a direct comparison, we considered only bug reports that included instructions for triggering the reported bugs through stress testing.

As listed in Table 6.8, the selected bugs exhibited different types of failures in various kernel components. Bug A causes a memory access violation (an "Oops" in Linux parlance) in the *pipe* communication mechanism, which can occur during concurrent *open* and *close* calls on anonymous pipes. Bug B also results in a memory access violation and is triggered on some interleavings when a FAT32-formatted partition is unmounted concurrently with the removal of an *inotify* watch[4] associated with the same partition. Bug C does not result in a crash, but rather causes a *read* system call to return corrupted values. Finally, bug D affects FreeBSD and is triggered by concurrent calls on sockets that cause the kernel to incorrectly return error values.

[4]Linux's *inotify* interface allows processes to receive change notifications for file system objects such as files, directories, or mount points.

Based on these four bug reports, we determined the system calls that would expose the bugs and produced the corresponding SKI test cases, as described in Section 6.5.3. For the bugs that had semantic manifestations, i.e., system calls that returned wrong results, we implemented bug-specific detectors, according to the information provided in the bug reports.

SKI exposed bugs A and B by triggering the crash after exploring 28 and 53 schedules, respectively. Bugs C and D were exposed after 51 and 3519 schedules, respectively, causing wrong results to be returned. Given that SKI requires few executions to trigger concurrency bugs, with a suitable test suite (e.g. regression test suites [GHK+01]), SKI's throughput is sufficient to reproduce on the order of hundreds of such concurrency bugs per hour (Table 6.1).

These experiments confirm that SKI is effective at reproducing real-world concurrency bugs. Most importantly, it should be noted that the reproduced bugs stem from two different OS code bases (FreeBSD and Linux) and from a wide range of Linux kernel versions spanning several years of intense development. In fact, even if we ignore the cumulative number of lines changed (i.e., the churn rate) and take into consideration only the increase in the total number of lines of source code, the Linux kernel grew by an impressive 60% from version 2.6.28 (10M SLOC) to version 3.6.1 (16M SLOC). SKI handled the different versions of the Linux kernel and the FreeBSD kernel without requiring any changes to the VMM itself or its configuration, which provides evidence for the considerable versatility intrinsic to SKI's design.

Comparison with stress testing

In the discussions that led to the resolution of these four bugs, the kernel developers proposed non-systematic methods to reproduce them. In particular, they provided simple stress tests, which continuously execute the same operations in a tight loop, waiting until a buggy interleaving occurs. We executed the original stress tests proposed by the developers to compare SKI to a traditional approach. For this purpose, we ran the stress tests in an unmodified VMM, i.e., without making use of SKI.

Note that without a deep knowledge of the kernel code, in the general case, it is hard to generate stress tests for the bugs that SKI discovered in Section 6.6.3. The reason for this is that it is not straightforward to ensure that, for every one of the various iterations of the stress test, the state of the kernel is such that it can trigger the concurrency bugs. (SKI avoids this problem because it automatically restores the initial state through snapshotting). Thus, to ensure a more objective comparison between the

Bug	Throughput
A	302.0
B	169.3
C	218.7
D	501.4

Table 6.1: SKI's throughput for each machine. Throughput is presented in thousand executions per hour.

two approaches, we chose to use stress tests produced by the kernel developers themselves since these are the ones offering better effectiveness guarantees.

As expected, and consistent with earlier comparisons of systematic and unsystematic user-mode concurrency testing approaches [MQB+08, BKMN10], SKI proved to be much more effective in reproducing concurrency bugs than the non-systematic approaches. Despite the fact that we gave each stress test up to 24 hours to complete, bug A and bug D were not triggered at all by their corresponding stress tests. While the stress tests for bugs B and C did eventually trigger their corresponding bugs, they required significantly more executions (and time) than SKI: the stress tests required more than 200,000 iterations (4 hours) to reproduce bug B and more than 800 iterations (1 minute) to trigger bug C, compared to 53 and 51 iterations (both a few seconds), respectively, under SKI.

Overall, the relative difficulty of reproducing bugs with simple stress tests is not surprising given prior comparisons of systematic approaches and stress testing in the context of user-mode applications [BKMN10]. Furthermore, this difficulty was also reported by the kernel developers themselves. For example, in the case of bug A (which the stress test failed to reproduce in our experiments) the developer stated that the "failure window is quite small" [BA2] and recommended introducing a carefully placed sleep statement in the kernel to trigger the bug.

Liveness heuristics

We instrumented SKI to log the activation of SKI's heuristics. Using this data we calculated the percentage of schedules that triggered each of the heuristics (Table 6.2) and the average number of times each heuristic was triggered per schedule (Table 6.3).

The results show that some of the schedules do not trigger heuristics. This is expected to happen when SKI chooses schedules in which threads do not experience lock contention and is more likely to occur in operating systems that are well optimized for

Bug	H1	H2	H3	H4	H*
A	1.72%	0.61%	5.71%	0.57%	7.97%
B	88.80%	49.70%	0.05%	13.73%	88.93%
C	1.50%	23.56%	0.00%	0.00%	25.06%
D	0.53%	2.66%	0.00%	0.00%	3.05%

Table 6.2: Percentage of schedules that triggered the liveness heuristics. H* refers to the percentage of schedules that trigger any heuristic.

Bug	H1	H2	H3	H4	H*
A	0.08	0.01	0.06	0.01	0.17
B	14.97	1.59	36.38	0.14	53.08
C	0.01	0.44	0.00	0.00	0.45
D	0.01	0.03	0.00	0.00	0.03

Table 6.3: Average number of times the liveness heuristics were triggered per schedule. H* refers to the percentage of schedules that trigger any heuristic.

scalability. Even though not all of the tests activate all heuristics, all heuristics were activated in at least one of the test cases.

In addition, we observed that in these tests the heuristics were triggered at 167 distinct instruction addresses. The large number of distinct addresses is indicative of the challenges that would result from manually annotating the kernel to infer thread liveness, as opposed to relying on the four simple heuristics implemented by SKI.

Effectiveness of communication points

To evaluate the effectiveness of the optimization of keeping track of communication points and allowing reschedules to occur only at these points (described in Section 6.4), we calculated for each test case the average number of instructions and the average number of communication points executed per run. As shown in Table 6.4, this optimization reduced the number of potential reschedule points by up to an order of magnitude in our experiments, thereby avoiding the wasteful exploration of redundant, effectively equivalent schedules. These results show the importance of this optimization to the scalability of SKI.

6.6.3 Exposing unknown concurrency bugs

To demonstrate the effectiveness of SKI in finding real world concurrency bugs, we tested several file systems from recent versions of the Linux kernel.

Bug	I	CP	I/CP
A	87673.5	12511.2	7.00
B	210693.0	23432.8	8.99
C	65126.9	6372.3	10.22
D	22641.3	6503.2	3.48

Table 6.4: Effectiveness of the communication points optimization described in Section 6.4. The table shows for each reproduced bug the average number of instructions executed per run (I) and the average communication points executed per run (CP). The last column characterizes the optimization's effectiveness as the ratio of the two metrics.

To create the inputs that form the various tests, we modified *fsstress* [Lar02], adding calls to SKI's hypercalls to flag the beginning and the end of the tests, and we modified the test suite to issue concurrent system calls. For convenience we also converted some of the debugging messages to use SKI's own debugging hypercalls. Because one of the file systems (*btrfs*) supports several operations that were not supported by the original *fsstress*, we also added support for twelve of those file system operations (e.g., snapshot/sub-volume operations and dynamic addition/removal of devices). In total, we added or modified 900 lines of code in *fsstress*, of which 700 lines are related to the *btrfs* operations.

Bug detectors

We ran SKI with three bug detectors. The first detector simply monitors the console output to detect crashes, assertion violations and kernel warning messages. We implemented this bug detector by configuring the kernel to redirect the console output to a serial device and by storing and analyzing its contents on the host. Despite its simplicity, in addition to flagging the executions that potentially triggered concurrency bugs, this bug detector can provide very useful kernel-specific diagnosis information, such as stack traces, warning messages and error messages.

The second detector uses file system checkers (*fsck*), which are specific to each file system and are only supported/mature in the case of some file systems, to detect file system corruption. This type of bug detector runs inside the VM, in contrast with the others, which are implemented within the VMM. To limit the performance impact of running *fsck* after each execution, we created small file systems (300 MB) and we mounted the file system in memory using *loop* + *tmpfs* (in addition to leveraging the optimizations described in Section 6.5.4).

Bug	Kernel	FS	Function	Detector / Failure	E	FS	Status
1	3.11.1	Btrfs	btrfs_find_all_root()	Crash: Null-pointer	41	0.030	Fixed
2	3.11.1	Btrfs	run_clustered_refs()	Crash: Null-pointer + Warning	26	0.020	Fixed
3	3.11.1	Btrfs	record_one_backref()	Warning	74	0.030	Fixed
4	3.11.1	Btrfs	NA	Fsck: Refs. not found	11	0.200	Rep.
5	3.12.2+p	Btrfs	btrfs_find_all_root()	Crash: Null pointer	61	0.060	Fixed
6	3.12.2	Btrfs	inode_tree_add()	Warning	53	0.010	Fixed
7	3.13.5	Logfs	indirect_write_alias()	Crash: Null pointer	31	0.065	Rep.
8	3.13.5	Logfs	btree_write_alias()	Crash: Invalid paging	142	0.020	Rep.
9	3.13.5	Jfs	lbmIODone()	Crash: Assertion	74	0.005	Rep.
10	3.13.5	Ext4	ext4_do_update_inode()	Data race	32	0.005	Fixed
11	3.13.5	VFS	generic_fillattr()	Data race	125	0.005	Rep.

Table 6.5: Bugs that have been discovered by SKI in recent versions of the Linux kernel and that we have reported to developers. For the specific input that triggered each bug, we show the number of schedules that were required to expose the bug (E) and the fraction of schedules that triggered the bug (FS). Eventually we found out that bug #3 had previously been reported. A patched version of the kernel, expected to solve bug #1, was tested on request from the developers but SKI revealed that the kernel could still crash in a different location of the same function (bug #5).

The third bug detector consists of a data race detector that we implemented, which analyzes all memory accesses, without sampling. Similarly to other data race detectors [EMBO10], our detector finds *racing memory accesses* without distinguishing whether those accesses are performed by synchronization functions. The main challenge in this case is filtering out the false positives [EMBO10, SBN$^+$97, MMN09, YRC05]. False positives can be of two classes: *false* data races (memory accesses that do not constitute data races) and *benign* data races (memory accesses that constitute data races but that are not considered harmful by its developers).

		Reports
False data race		76
Data race	Benign	90
	Harmful	24

Table 6.6: Types of race reports found during our experiments. The numbers displayed refer to the number of reports after associating related races. Note that a single bug may be involved in multiple data races (e.g., if it affects multiple variables).

	Btrfs	Ext4	Jfs	Logfs
SKI	34.7	62.6	61.6	61.2
SKI+ DR	32.1	61.9	59.5	58.8
SKI+ Fsck	6.4	20.8	18.2	N/A
SKI+ Fsck + DR	6.1	20.6	17.9	N/A

Table 6.7: SKI's throughput (for each machine) with different bug detectors. Throughput is given in thousands of executions per hour. DR denotes the data race detector. Fsck tests on *logfs* are absent due to the lack of compatible mature checkers.

Bug	Kernel	OS Component	Failure	E	FS
A [BA]	Linux 2.6.28	Anonymous pipes	Crash	28	0.00572
B [BB]	Linux 3.2	Inotify + FAT32	Crash	53	0.13770
C [BC]	Linux 3.6.1	Proc file system + Ext4	Semantic	51	0.01004
D [BD]	FreeBSD 8.0	Sockets	Semantic	3519	0.00014

Table 6.8: Known bugs reproduced with SKI. The table shows the number of schedules that were required to expose the bug (E) and the fraction of schedules that triggered the bug (FS). The table shows the kernel version under which we reproduced the bug, the OS components involved and the type of failure that the bug causes.

In order to facilitate the manual process of analyzing the false positives produced by the data race detector, our tool groups together distinct pairs or racing instructions that were found to race directly or transitively. Using this method, we were able to group together 3114 pairs of races into 190 race reports. Filtering out race reports that were not data races was straightforward – it consisted of ruling out the races that occur within the synchronization mechanisms, which we accomplished using the kernel debugging symbols that map instructions to source code locations. In contrast, the difficult part was separating real data races into benign and harmful ones. In some cases, this process requires careful analysis of the code and documentation and, ultimately, it may require asking the developers – who may not even agree among themselves. Heuristics could have been used to analyze the results, but unfortunately these typically offer limited help for the more complicated cases. Given this complexity, we gathered some reports (not included in Table 6.5) that may constitute bugs but are still under analysis, and for which, in some cases, we are still waiting for feedback from the developers. Table 6.6 shows the number of race reports that we obtained in the file systems tests according to their type.

Results

The results in Figure 6.5 show that SKI was able to find several unknown concurrency bugs in mature versions of the Linux kernel. One of the bugs found affects the widely used *ext4* file system and six bugs affect the *btrfs* file system – which is expected to soon become the default file system in some distributions [OPE]. We have reported the 11 bugs listed in Table 6.5; of those, 6 have already been fixed.

Furthermore, although FS related system calls tend to be expensive, SKI was able to achieve a testing throughput that reached 62 thousand executions per hour on each machine (Table 6.7). Even though the current performance of SKI proved to be effective, significant performance improvements may still be achievable by using more efficient virtual machines monitors, possibly using hardware acceleration, or even by building SKI using binary instrumentation frameworks.

It is worth pointing out that many of the bugs found by SKI are serious – six of the bugs cause the kernel to crash and most of the bugs found cause persistent data loss. For example, the *ext4* bug, which is due to improper synchronization while updating the inodes, causes the field *i_disksize* (containing information about the size of the inodes) to become corrupted. To fix this bug, developers applied patches that involved refactoring the code and the introduction of additional synchronization.

6.7 Limitations and discussion

6.7.1 Reliance on thread pinning

SKI establishes a one-to-one mapping between threads and virtual CPUs by pinning threads, however, this technique has two limitations. First, the process of pinning threads is restricted to certain types of threads. Second, pinning threads imposes restrictions on the kernel execution paths that can be tested. This section discusses both of these restrictions and their implications on testing kernels using SKI's approach.

Applications cause kernel code to be executed by invoking system calls in user-space or, similarly, by triggering exceptions (e.g., divisions by zero, page faults). Given that system calls are an important entry point into kernel code and often the mechanism used to drive the kernel during testing [Lar02], the work presented in the previous sections focused on demonstrating the effectiveness of SKI when testing kernels through the invocation of system calls. Nevertheless, not all kernel code is directly executed through the invocation of system calls by applications.

In addition to applications, *kernel threads* [KER] are also responsible for executing kernel code. Kernel threads are a thread abstraction, implemented at the kernel level, that does not have a user-mode component associated[5]. This mechanism is used by kernel programmers to asynchronously execute tasks that perform computations on the internal state of the kernel (without the direct involvement of applications). For example, in Linux the *pdflush* kernel thread is responsible for managing the page cache and the *ksoftirqd* kernel thread is responsible for part of the interrupt handling under high-load situations.

Despite not being associated with user-mode processes, kernel threads can still, in many cases, be pinned to different CPUs. Pinning kernel threads simply requires that an application, constituting a component of the testing infrastructure that runs within the virtual machine, invokes the thread affinity system call and identifies the target kernel thread using its process identifier (PID).

However, there are cases of kernel threads that cannot be pinned to arbitrary CPUs for OS-specific reasons. For example, in Linux a *ksoftirq* kernel thread is spawn for each one of the CPUs and is pinned within the kernel itself to its respective CPU, which has the advantage, from the kernel programmers point-of-view, that it is possible to rely exclusively on CPU-local variables to maintain the state of these threads, and, therefore, the synchronization of the accesses to its state is simpler. For the subset of kernel threads that cannot be pinned to arbitrary CPUs, SKI is not able to systematically control their schedule and, therefore, has to leave the schedule control to the native kernel scheduler.

Another consequence of pinning the tested threads to CPUs is that it restricts the kernel functionalities that can be tested with SKI. As an example, SKIis not able to test the scheduler code responsible for migrating threads between CPUs because migration is necessarily prevented when threads are pinned to a single CPU. In essence, the requirement to pin the tested threads constitutes a limitation on the input space that can be tested with SKI, namely by preventing tests from considering an arbitrary affinity set for the tested threads, which consequentially restrict the testable execution paths. However, we expect this limitation to have a minor impact on the overall applicability of the proposed testing approach, which would mostly affect the scheduler logic, because the kernel scheduler constitutes a relatively small proportion of the entire kernel code. Therefore, the proportion of bugs for which this approach is not be applicable is expected to be similarly small.

[5]The *kernel thread* term is used with a different meaning in other work [McC02] to distinguish a thread abstraction implemented at the user-space level (*user thread*) from a thread abstraction implemented at the kernel-space level.

6.7.2 Reliance on VMM

SKI proposes a VMM-based scheduler. In this section, we discuss some of the implications of this choice.

A limitation of relying on a VMM is that the kernel running inside a virtual machine is limited to using the hardware virtualized by the VMM. For a testing tool, it means that it is not possible to reproduce bugs that require hardware that is not virtualized by the VMM. In the case of our implementation, this problem is partially mitigated by the fact that, in comparison with other VMMs, QEMU supports an unusually large number of devices (over 250). Nonetheless, SKI cannot be used to diagnose concurrency bugs in drivers for devices not virtualized by QEMU. However, we believe this does not detract significantly from SKI's practical value because the size of the device-independent kernel core is already considerable. Further, it may be possible to overcome the VMM dependency by building an equivalent tool based on kernel binary instrumentation, which is an active area of research [FBG12].

The choice of a VMM-based approach has another important consequence. Because the VMM emulates one instruction at a time, and propagates its effects to all other CPUs immediately afterwards, concurrency bugs that arise from wrongly assuming a strong memory model are not necessarily exposed. This is because some CPUs offer weaker memory models, which can have very complex semantics, to the point where official specifications have been found to not match the observed semantics of hardware [AFI+08]. This is a complex problem — significant effort has been directed at simply studying the semantics of CPUs with relaxed memory models [SSN+09] — and we believe that effectively diagnosing this type of concurrency bug will likely require more specialized tools. Such bugs are currently not the target of SKI.

6.8 Summary

This chapter introduced SKI, the first practical testing tool to systematically explore the interleaving space of real-world kernel code. SKI does not require any modifications to tested kernels, nor does it require knowledge of the semantics of any kernel synchronization primitives. We detailed key optimizations that make SKI scale to real-world code, and we have shown that SKI is effective at finding buggy schedules in both FreeBSD and various versions of the Linux kernel, without changing or annotating the tested kernel.

7 Testing software in the real-world

This section takes a step back to analyze several important aspects of testing tools by taking into consideration our experience designing, developing and evaluating PIKE and SKI.

Section 7.1 enumerates and discusses the different properties of testing tools and Section 7.2 discusses the different classes of potential users of testing tools. Section 7.3 reports our experience interacting with software developers. Section 7.4 analyzes the problem of scalability in the context of testing concurrency bugs and, finally, Section 7.5 provides a reflection on the limits and future of software testing.

7.1 Properties of testing tools

There are several properties that combined determine the value of testing tools. We have identified four important classes of properties (Table 7.1) that should be taken into consideration both by developers, when deciding which tools to adopt, and by researchers, when designing tools. Unfortunately, since testing tools have many different dimensions, with regard to their properties, there does not seem to be a single *best testing tool*, in fact, developers can (and do) benefit from using simultaneously several testing tools that complement each other.

7.1.1 Scope

Bugs tested. The scope of bugs that are testable by a given testing tool is a property generally inherent to its design. The scope of bugs testable refers to several aspects, such as their external effects (crash, semantic, hang) and their internal effects (latent, non-latent). In addition, testing tools can have restrictions with regard to the location of the bugs within the code; for example, certain portions of the kernel scheduler code cannot be tested with SKI (Section 6.7). Testing tools may additionally have more subtle restrictions, namely regarding the type of concurrency considered, for instance, SKI considers instructions to be executed atomically and, therefore, only exposes concurrency bugs that do not depend on weak memory models (Section 6.7).

Properties
1. Scope
Bugs
Software
2. Manual effort
Setup
Analysis
3. Performance
4. Hardware requirements

Table 7.1: Summary of the properties of testing tools.

Software tested. Applicability, in terms of the type of software that can be tested, is another important aspect of testing tools. For instance, certain testing tools are geared towards testing kernels (e.g., SKI), others are geared towards testing user-mode applications, yet others are applicable to both types of software. Testing tools can also have limitations in terms of the programming languages supported, especially if static analysis techniques or source-code level instrumentation is leveraged.

7.1.2 Manual effort

Setup. Setup costs can arise from different sources. Certain testing tools, such as PIKE, require developers to modify or to annotate the tested software. Such requirement constitutes a cost per tested software and, potentially, a cost per software update, to maintain the modifications or annotations. In practice, the cost of modifying the tested software depends strongly on the required modifications and can be significantly mitigated by building reusable libraries that reduce the complexity of implementing such modifications.

Dynamic testing tools, which depend on the execution of the tested software, have another type of setup cost – test cases need to be provided to steer the execution of the software during testing. In general, it is considered a good practice to write test cases [Viz07], regardless of which testing tool is used, so the dependence on test cases does not necessarily translate into an extra burden for the developers because it may be possible to simply reuse (or adapt) pre-existing test cases.

While conducting the work presented in this dissertation we had the opportunity to interact with developers, who stressed the importance of the usability of the tool. An aspect that developers attributed significant importance was the ease of creation of test cases and the quality of the documentation. In fact, developers explicitly mentioned that testing tools should provide template test-cases and a clear methodology to diagnose problems, in case anomalies occur during testing due to user error (e.g., constructing incomplete test cases or providing incompatible testing parameters).

Analysis. The quality of the results determines the effort that users have to invest to extract useful information. Testing tools that provide reports with detailed information allow developers to easily diagnose the causes of bugs and to implement a fix. For example, the test cases that we have built with SKI, in addition to exploring systematically the interleaving space, have the advantage of reporting exactly which were the system calls invoked concurrently and their parameters. Furthermore, SKI is able to provide developers with the exact memory state of the machine before the system calls were invoked and detailed execution traces. Stress tests on the other hand generally provide significantly less information – in fact, given the inherent lack of control over the interleavings executed, enabling precise logging can interfere with the interleavings execute by stress tests and, therefore, potentially reduce their effectiveness.

Another factor that determines the quality of the results produced by testing tools is the frequency of false positives – certain types of testing tools are prone to producing spurious bug reports in cases where the software is correct. False positives can become a serious problem if they overwhelm the user or if they hurt the confidence developers have on the testing tool. Certain types of tools are likely to produce false positives by design – data race detectors are one example – but even in these cases, testing tools may include mechanisms that mitigate the problem by leveraging heuristics to rule out the cases that are likely to be false positives.

In addition to the amount of information available in the bug reports and its accuracy, the form of the results is also important, albeit in a more subtle manner – developers have a preference for reports that use a familiar format. This preference becomes more relevant in collaborative development environments, in which developers have to interact with other developers to diagnose problems. In such common environments, having a familiar report format avoids the need to educate many developers with regard to the format of the results (which could also require explaining the inner workings of the testing tool). This fact motivated our decision to implement the crash detector in SKI by monitoring the console output and by capturing the bug reports produced by the standard kernel methods (as opposed to detecting the execution of certain instructions or the occurrence of architecture-level exceptions).

Another important aspect, to ease the process of analysis of results and bug fixing, is the reproducibility of results. Reproducibility allows developers to run tests repeatedly to rule out transient external causes (e.g. a faulty machine) and allows different developers to witness the same results (in their own testing environment). But even more useful is the ability to achieve a certain level of stability of the results, i.e., reproduce tests even if the software has been (slightly) modified. Tools that have this property allow developers to modify the tested applications, perhaps in subtle ways, to explore different hypothesis regarding the cause of the problems under analysis or to acquire additional debugging information, using instrumentation.

7.1.3 Performance

The time it takes for the test results to be returned is important to developers – faster tools encourage frequent testing and allow developers to find bugs earlier, which is well known to reduce the fixing cost [BP88].

The speed of testing is dependent on the testing tool used, as well as on the computational resources available (e.g., CPU speed and number of cores). However, testing tools that are parallelizable (such as PIKE and SKI) are more flexible with regard to the tradeoff between speed and hardware resources available – parallelizable tools are able to produce results faster if more cores or machines (or faster machines) are provided, whereas tools that are not parallelizable are bounded by the CPU speed of the machine used.

7.1.4 Computational resources

The computational resources required by testing tools varies widely. For example, some static tools require few computational resources and do not need more than a single machine to produce results within a few seconds. On the other end of the spectrum, dynamic testing tools generally have higher computational costs and potentially require large testing infrastructures supported by several testing machines to produce results within hours or days. Obviously, the availability of computational resource has an impact on the performance of the testing tools (Section 7.1.3).

Approaches that are on the higher end of the spectrum, in terms of computational resource requirements, are meant to be used differently than less demanding approaches. Less demanding approaches can be integrated into the compilation phase, during development, being able to test the private working version of the software of each individual developer. In contrast, more demanding approaches might be too expensive to be used in this situation and be meant to run tests in the background, by testing a snapshot of the latest version of the software (within the company) or even the deployed version.

User class	FS	FT	EC
Developers of new features	High	Low	High
Internal dedicated testers	Medium	High	Medium
External dedicated testers	Low	High	Low
Software end-users	Low	Low	Low
Researchers	Low	High	Low

Table 7.2: Summary of the profiles of different classes of users. The table presents for each user class the expected familiarity with the tested software (FS), familiarity with the testing tool (FT) and ease of communication with the developers (EC).

In addition to the amount of computational resources required, testing tools may require dedicated hardware. For example, dynamic tools may require testing machines with specific CPU features or architectures (e.g., DataCollider leverages hardware watchpoints), while dynamic tools based on virtual machines, like SKI, tend to be more flexible with regard to the physical CPU. Researchers have also developed tools that leverage and require, for functional or performance reasons, GPUs [RSSK14, DZW^{+}14] and FGPAs [TER].

7.2 Users of testing tools

For presentation purposes, the previous sections refer to the users of testing tools using the generic term *developer*. In practice, however, the individuals responsible for developing software and incorporating new software features are not the only possible users of testing tools. Table 7.2 provides a summary of the profiles of different user classes and the rest of this section analyzes the importance of different features for each user class.

7.2.1 Developers of new features

Software developers responsible for implementing new software features are an obvious class of users of testing tools. This class of users is part of the development team and, therefore, has in-depth knowledge of the tested software. For this reason, tasks that involve modifying the tested software or gathering information about the semantics of tested software are easier for this class of users than for other classes of users (as discussed later in this section). As a consequence, this class of users is in a good position to implement the state abstraction functions that PIKE requires.

However, this class of users is not dedicated to testing and often has other pressing tasks to perform, namely implementing new software features. Therefore, the interface of the testing tool should be particularly easy to learn and use, as this class of users cannot easily amortize the learning costs.

7.2.2 Internal dedicated testers

Some software development organizations have teams responsible for testing software that can leverage different types of testing tools. These teams are not meant to replace basic testing conducted by software developers, instead they are meant to complement them. This class of users is different from the developers class because it is less familiar with the code being tested, in particular, when problems are suspected these users may need to communicate with the developers to confirm the problems. On the positive side, this class of users has the advantage that it is dedicated to testing and so can afford to learn the inner workings of different testing tools and might be able to spend additional effort fine tuning them to achieve higher effectiveness.

7.2.3 External dedicated testers

Organizations specialized in testing (e.g., Coverity [COV]) that offer their testing services to software development organizations are another potential class of users. Likewise, testers in the context of crowd-testing (crowd-sourced testing [uTe, LLKB12]) – essentially, individuals (presumably proficient at testing) with a diversified background that offer their testing services to other organizations – are also part of this potential class of users. In practice, in both cases software developers outsource part of their testing responsibility to external individuals or organizations.

Because this class of users is dedicated to testing, users are expected to be familiar with different testing methodologies and, similarly to the *internal dedicated testers*, are expected to be more willing to dedicate a high amount of effort to adopt novel testing approaches. However, in contrast with users from the software developing organization, external testers are expected to have less knowledge about the internals of the tested software and the communication barrier with the software developers is higher.

7.2.4 Software end-users

Software end-users can also be users of testing tools. This scenario is more likely to occur in the context of open-source projects (since testing becomes harder without access to source code) and when the end-users are tech-savvy individuals or organizations with significant interest in the software.

End-users are generally neither proficient at testing nor familiar with the internals of the software. However, especially in the case of software systems that are popular, end users can provide important feedback to developers, even if just by performing simple forms of diagnosis, given the sheer number of individuals and the quantity of information they can collect.

7.2.5 Researchers of testing tools

The researchers (and developers) of testing tools are also users of testing tools. Usually their immediate primary motivation for using testing tools is to validate their design and indirectly, in the long-term, to improve the robustness of software. Given their role, it is expected that this class of users is familiar with the internals of the testing tools but less familiar with the software tested. Given their lack of familiarity with the tested software, this class of users can serve to give an estimate on the upper bound of the effort required to make application-specific changes for testing purposes, such as those we performed to MySQL when validating PIKE.

7.2.6 Discussion

The classes of users that we discussed in this section can work collaboratively for the common goal of testing a given software system. In fact, a compelling usage scenario, for both PIKE and SKI, is to have a shared testing infrastructure, maintained by a few dedicated testers (either internal or external), with test cases submitted by all classes of users. This model permits all classes of users to contribute to test the software and is a model followed, for instance, by the Linux Test Project [Lar02]. The potential disadvantages of this model is that it requires a certain coordination effort and, in some cases, users might not be comfortable sharing test cases, due to internal organizational policies or business secrecy concerns.

7.3 Experience interacting with potential users

This section reports our experience interacting with potential users in the process of reproducing known bugs (Section 7.3.1) and reporting bugs found by our testing tools (Section 7.3.2). In addition, Section 7.3.3 discusses the feedback we got from potential users regarding the design of SKI. We believe that most of the individuals we interacted with were software developers responsible for building new features into the software who also had testing responsibilities.

7.3.1 Reproducing bugs

One of the approaches we used to evaluate SKI consisted in reproducing already known bugs in old versions of the kernels (Section 6.6.2). This requires creating SKI test cases that provide a particular input to the tested kernel, by invoking certain system calls with certain parameters, to the tested kernel. However, obtaining all the necessary information by simply analyzing bug reports, which often do not state the required input explicitly, proved to be a difficult task. For instance, many crash bug reports included a stack trace that was sufficient to infer one of the system calls, but did not specify which are the other system calls that need to be executed concurrently. Likewise, the exact system call parameters, the initial state of kernel, the kernel version, the build configuration and the run-time configuration were in many cases not explicitly stated in the bug reports. In fact, reproducing bugs using information from bug reports, regardless of being concurrency bugs or not, is a challenging problem (orthogonal to the problems targeted by our proposed tools), which has received the attention of other researchers [VFS14, BPSZ10, GZNM11].

Our initial strategy to find the missing information from the bug reports resorted to asking the developers, involved in the bug diagnosis process, to provide us additional information. Unfortunately this strategy didn't work well – we were able to engage in conversations with the developers but, due to several factors, we were not able to acquire significant information using this strategy. In several cases the bugs were excessively old (i.e., months or years) to the point where developers could not easily recollect the details. We believe that the fact that many of the concurrency bugs were considerably complex also contributed to this problem, by making it harder for developers to recall the intricate details involved and also by requiring more time from the (already busy) developers to reanalyze the bugs. Curiously, in one instance, a developer informed us that he had a constraint of a different nature – the developer was not allowed to provide additional information to reproduce a concurrency bug, due to internal organizational policies, because it was considered a security bug and it could be triggered by any OS user without special privileges.

Given these challenges, we finally opted for the strategy of limiting our evaluation to bugs that had very well documented bug reports and that included stress tests. This strategy proved to be successful – we were able to reproduce several concurrency bugs with the stress tests and, subsequently, we were able to confirm that SKI is able to effectively reproduced them (Section 6.6.2). Taking into consideration the challenges involved in analyzing bug reports and the limited value of reproducing already known (and fixed) bugs, we focused most of the evaluation effort in finding unknown bugs.

7.3.2 Reporting bugs

We reported the concurrency bugs found by SKI to the developers of the Linux kernel. All bugs found (Table 6.5) were confirmed by the developers to be concurrency bugs and more than half of these concurrency (6 out of 11 bugs) have since been fixed.

We observed that, in general, the bugs that were fixed pertain to components of the kernel that are currently under active development and that have active development teams, such as the BTRFS team. Conversely, bugs that were not fixed, in general, involved components that recently have seen less active development. Analyzing bug reports and fixing bugs is understandably a complex task that demands significant effort and time from developers, for this reason, it can be difficult to gather the attention of developers and ensure that bug reports receive a positive response that leads to a fix.

Our interaction with the developers is consistent with a correlation between the level of activity of the development teams affects and how likely it is to get confirmation from developers and their involvement in producing a patch. However, in this process we realized that there are other important factors that contribute to affect the chances of engaging with the developers and getting a positive response from them. These factors include the form of communication used (e.g., development mailing lists or directly with individual developers), the exact point of contact (e.g., which developer), the software version used, the external effects of the bug and the completeness of the bug report.

Throughout our evaluation process, we relied on different forms of communication with the developers, depending on the circumstances. Development mailing lists were used in several occasions to report and diagnose the concurrency bugs found and, in the absence of active mailing lists or in the absence of a response, we also contacted individual developers directly. When contacting developers directly, we favored developers that were officially assigned as maintainers of the component affected and the developers that last modified the kernel source code that was relevant for the respective bug (e.g., the crash or data race locations).

We also observed that bug reports pertaining to older versions of the software tend to receive less attention from the developers and, in such cases, the bug reporters may be asked to re-run the tests on the most recent versions. A few minor versions behind the latest released version may be sufficient to cast doubt on the report. For instance, MySQL reporting guidelines instruct bug reporters to verify that the reported bugs affect a product version that is no "more than two revisions older than the latest version" [Cor]. This requirement is justified by the fear that such bug reports might have become obsolete simply because the affected software functionality might have been modified or because the bug might have been fixed in the meantime.

Bug reports that have serious effects tend to easily capture the attention of developers. For example, crashes or situations that lead to data loss are generally more important to users than performance degradation bugs and, therefore, bug reports describing such situations are naturally prioritized by software developers. A related aspect that developers expect in bug reports is clear evidence that the reported situation can cause harm to users. For concurrency bugs, bug reports involving data races are a typical case that can raise doubts whether the reported situation actually constitutes a bug that actually causes harm to users.

Our experience analyzing and discussing situations involving data races with kernel developers showed that developers expect a clear reason to fix the code in order to avoid a data race. In fact, developers expect to have a concrete description of the process by which the suspected kernel bug can cause an application to *break*. Several reasons contribute to the conservative approach of developers when deciding whether a bug should be fixed, including the fact that the potential fixing strategy could increase the software complexity, slow down the software or potentially create more serious problems (if the fix is incorrectly implemented). Furthermore, organizational factors may also have an important role – in a collaborative development environment, such as the one of the Linux kernel, a developer that proposes a bug fix needs to be sufficiently confident about its correctness and needs to be able to justify his or her actions to other developers.

7.3.3 User feedback

We explicitly requested feedback from the kernel developers regarding the design of SKI and the problem of testing kernels for concurrency bugs. The developers we contacted confirmed that, using their current practices, testing for concurrency bugs is a challenging task and that stress testing is the best method they rely upon for this purpose. In addition, they also expressed dissatisfaction with regard to stress testing and explicitly mentioned that stress tests are not executed often because such tests can "take a day or two to run".

When informed of the development of SKI, the developers strongly expressed enthusiasm about it and manifested their desire to use SKI. Regarding the design, the developers emphasized the importance of the ease of use with respect to setup and execution, as well as the process of test case construction.

7.4 The scalability of testing concurrent software

Testing software systems for concurrency bugs, in the simplest case of testing two concurrent threads, can be interpreted as exploring a large three-dimensional space. According to this view, the testing space has two dimensions, which correspond to the input provided to each of the two tested threads, and a third dimension, which corresponds to the interleaving of instructions that is executed. As such, testing concurrent software essentially consists in executing the software, under different circumstances that correspond to different points in the space, and applying bug detectors that analyze each execution, to uncover potential violations of the intended semantics. Since, under the usage scenario we envision, software is expected to be reasonably well tested, the vast majority of the points in this three dimensional space is expected to represent executions that comply with the application specification, therefore, only a tiny fraction of the points in the testing space is expected to expose concurrency bugs.

In stark contrast, testing for non-concurrency bugs involves a dramatically smaller space, even though it is still challenging – testing for non-concurrency bugs can be interpreted as exploring a single dimensioned space that simply corresponds to the input request provided to the (single) thread being tested. In comparison with non-concurrency testing, the fact that concurrency bug testing has two extra dimensions, together with the fact that non-concurrency testing is already a very challenging task, means that the scalability of testing needs to be carefully addressed to ensure effectiveness.

Apart from improvements to the performance of the testing infrastructure, in this work we followed two high-level strategies to deal with the scalability concern. The first strategy consists of prioritizing, or limiting, the points explored by leveraging testing heuristics (Section 7.4.1), while the second strategy consists in analyzing several points from the testing space simultaneously (Section 7.4.2).

7.4.1 Prioritizing the testing space exploration

The strategy of giving priority to a subset of the testing space is motivated by the fact that, in general, for large software systems there is no expectation of exploring the entire space. Consequentially, it is particularly important to apply effectively the available testing resources by exploring first the points in the testing space that are more likely to yield positive results.

The tools that we propose in this dissertation follow this strategy by including several heuristics. Notably, with regard to the exploration of the interleaving dimension, both PIKE and SKI bound the number of reschedule points. Furthermore, both tools select the location of the reschedule points and the initial priorities of the execution contexts using uniformly random approaches. Another example of the implementation of this strategy arises in the context of the selection of the testing input – our experiments implicitly use heuristics by leveraging existing test suites that we converted into concurrent test suites. Test suites, such as those leveraged, implicitly encode heuristics by including tests that were created based on programmer intuition, taking into consideration his knowledge of the software, or based on previous bug reports, taking into consideration previous bug patterns.

7.4.2 Reasoning over multiple executions

The second strategy to improve the scalability of testing consists of exploring, or reasoning about, several points within the testing space simultaneously. In contrast with the previously discussed strategy, this strategy has the advantage of providing stronger guarantees by not ruling out the analysis of certain points of the testing space, but it can be harder to implement.

Our implementation of SKI analyzes the communication between different CPUs (Section 6.4) and, based on this information, SKI is able to conclude that certain classes of interleavings are equivalent and therefore only one member of each class needs to be executed. This technique allows SKI to significantly reduce the testing space with regard to the interleaving dimension (Section 6.6.2).

A prime example of an implementation of this strategy is the general class of symbolic execution techniques, discussed in more detail in Section 2.2.2. Symbolic execution techniques allow the testing infrastructure to reason about several concrete executions, that correspond to the same execution path, in a single symbolic execution. The typical source of concern regarding this approach is the management of the path explosion problem, for which several practical methods have been proposed. However, the strategy of reasoning over multiple executions is broader than symbolic testing, since it is not limited to the simultaneous execution of points that share the same execution path. In fact, it is sufficient if the bug detectors are capable of reasoning about several executions by analyzing a single one.

Although with limited gains, a race detector, such as the one that was leveraged in Section 6.5.5, serves to illustrate well how a bug detector can reason about multiple executions from a single execution. After a data race detector detects two racing instructions, in principle, it is no longer needed to execute the point in the testing space that corresponds to the alternative data race outcome to be able to conclude that those instructions race. Nevertheless, the execution of the alternative data race outcome might have an important value for bug diagnosis purposes (i.e., to allow the programmer to understand both outcomes and more easily conclude whether the effects of the data race are harmful or not).

7.4.3 Discussion

Despite the scalability benefits of the strategies discussed in the previous sections, testing concurrent applications remains computationally challenging. For example, testing a file system using the methodology that we describe in Section 6.6.3 can take more than a day on a testing infrastructure backed by a ten-machine cluster, even with our most optimized implementation of SKI. In addition to the computational cost, the development and configuration of the testing tools themselves also constitute a cost, in terms of developer time and effort. However, on the positive side, automated testing tools have the important advantage of freeing developers from doing manual analysis and constitute an opportunity to find bugs earlier in the development cycle, which can greatly contribute to the reliability of software. As such, this section discusses the cost-benefit ration of the automated testing techniques proposed in this dissertation.

In comparison with traditional non-automated testing approaches (e.g., manual analysis of application specific logs or collection of information from bug reports), automated approaches, like those proposed in this dissertation, enable developers to create testing infrastructures that, once they are setup, require only very limited interaction from developers.

This difference is important because it converts the labor cost, associated with testing, from a variable cost (manual analysis of each bug report or each tested software version) into a predominantly fixed cost (development and configuration of the testing infrastructure). Freeing developers, not only translates into saving an extremely valuable resource, but it also has the strategic advantage of encouraging systematic software testing. In addition, the diagnosis information provided by our tools (e.g., SKI produces detailed execution traces) is able to further reduce the labor costs with regard to analyzing and fixing concurrency bugs. It is not unusual for concurrency bugs to take weeks to diagnose, even after being reported by experienced users, despite the intense efforts and the competence of the developers that try to fix them [LOCa], so tools that provide detailed diagnosis information are an important asset to developers. Regarding the effective computational cost of our tools, it is also worth mentioning that this cost is lower than it may appear because our tools can run the tests opportunistically by using resources when they are not being used for other purposes (e.g., a cluster with low load).

The testing methodologies that we proposed in this dissertation also have two strategic advantages that derive from being embarrassingly parallel. First, being embarrassingly parallel allows tests to run faster in the future, regardless of whether hardware architects adopt a frequency scaling or a multicore scaling strategy. Second, this property enables users to run tests (virtually) arbitrarily fast, assuming that enough machines are available, thus increasing the opportunities to integrate the testing methodology into the development cycle of programmers (e.g., allowing tests on new functionalities to finish in time to apply the fixes and before the software is released to clients).

7.5 Towards reliable software

Writing software, by developers using programming languages, is comparable to writing text, by writers using natural languages[1]. In both cases, humans have a task that requires them to encode information, although using different types of languages, and in both cases the authors want their artifacts to be defect free.

Because both types of tasks are very challenging, several automated approaches have been developed to aid the authors in both contexts. In the case of text writing, word processors often include spelling or grammar checkers and other tools to check the formatting of documents (an example of the later are the tools used by conference manuscript submission websites to check compliance with formatting guidelines). Similarly, many tools have been developed to help software developers. Examples range from the errors and warnings produced by compilers to more complex tools, like those proposed in this dissertation.

The paramount complexity of ensuring the correctness of artifacts becomes clear when considering the extremely high degree of freedom that authors have and that any producible artifact is potentially the one (and only) that the author intended. Consequentially, an *ideal* testing tool – meaning a tool capable of automatically detecting all bugs – needs to be able to distinguish any two, even if seemingly valid, implementations and automatically conclude which one of them is intended by the developer. For example, in the extreme case, a developer could end up writing an artifact, such a kernel, when it instead intended to write a completely different artifact, such as a database manager. In this scenario, an ideal testing tool would have to detect the (gross) mistake of the developer and, unfortunately, achieving this formidable goal amounts to building a tool that guesses the developer's intention to an extent not foreseeable.

[1] It is even arguable that books and software are equivalent because any software can be encoded in a book (e.g., by printing its source code [Zim95]) and likewise any book can be encoded in software (e.g., using e-book readers [Gol08])

Verification [Hol97] and program synthesis [MW71] approaches, despite providing strong guarantees, are also unable to bypass the challenges that the ideal testing tool would have to overcome. These approaches typically provide guarantees about software with regard to a specification, but a correct specification still needs to be written by someone. In essence, even the existing tools that provide the strongest guarantees can not overcome the fact that whether or not a certain aspect of the software is a bug ultimately depends on the intentions of the developers.

In practice, existing approaches to improve the reliability of software rely on two methods to circumvent, with some compromises, this challenge. The first method, which is used by verification and synthesis tools, consists in explicitly asking developers for information regarding what constitutes correct behavior. This is achieved by requiring developers to write software, or part of it, either in a higher-level language (e.g., a specification) or otherwise in a redundant manner (e.g., n-version programming [AC77]).

The second method, relies on encoding heuristics about the correctness of software within the tool. This method relies on patterns of correctness that apply to a broad range of software. A simple example is the heuristic used by crash detectors – it is virtually universal that programmers do not want their software to crash. PIKE, however, uses a hybrid approach by combining both methods – PIKE leverages application-specific information to capture the state, which is achieved through the use of annotations, and additionally encodes a heuristic that assumes that correct applications generally have linearizable semantics.

However, even if the ideal testing tool existed, ensuring end-to-end reliability would still require ensuring the correctness of hardware (both at the design and manufacturing levels). This is an equally challenging problem (if not more) – ensuring hardware correctness is particularly hard given that modern hardware has become extremely complex and is, in some cases, expected to operate under extreme conditions (e.g., high frequency, high temperature, low voltage). Hardware design verification techniques have made important progress, but manufacturing defects can still occur and cause serious problems [INTa, KK98].

Recognizing the magnitude of the software reliability problem should not discourage researchers and developers from doing better. We believe there is significant progress that can be made with regard to testing tools, namely by improving the properties of the testing tools listed in Section 7.1, and that these improvements will have an important impact on the reliability of our computer systems. The following chapter discusses in more detail several research directions that directly follow-up from the work presented in this dissertation.

8 Future research directions

This chapter discusses three main research directions that follow-up from the work presented in this dissertation.

8.1 Practical extraction of application state using its API

PIKE in its current form requires application-specific annotations to test applications. This requirement forces developers to dedicate some effort to be able to use the tool, which is a disadvantage. We believe it may be possible to overcome this limitation by refining the design of PIKE.

Nevertheless, it is worth noting that the impact of the specific annotations required by PIKE's current design is mitigated by several factors.[1] First, the modifications required by PIKE do not need to modify the state of the application and, consequently, annotations incorrectly implemented are unlikely to introduce new bugs in the application. In fact, this concern could even be entirely avoided by isolating the annotations, for example, using the *ptrace* mechanism to gather the state of the tested application. Instead, mistakes in the annotations would have an impact limited to the state summaries produced. Second, the annotations can be implemented selectively and incrementally, which allows the developer to control the tradeoff between manual effort and testing scope. Third, our experience showed that the approach taken by PIKE can be effective despite a limited amount of programmer effort and despite the fact that we considered a particularly complex type of application (a database manager) that we were not as familiar with.

Regardless of the mitigating factors of this limitation, an interesting research direction consists in exploring refined designs of PIKE that forgo the need to provide application-specific annotations. More specifically, gathering the summary of the state of the application may be possible by leveraging, exclusively, the application API. For example, to test for *query cache* latent bugs, PIKE could simply invoke at the end of the test

[1]In the context of the alternative design considered for SKI (Section 6.3.1), because none of these factors apply, we ultimately rejected the alternative design and, instead, proposed an entirely transparent design.

SELECT requests that match the requests that were previously invoked concurrently, to test whether the cache results are wrong. Another example would be to use administrative SQL statements, such as SHOW requests, to expose information about the application state that is not made externally visible by non-administrative requests.

The external API approach would still require some testing effort, but would have several important advantages in comparison with using annotations that rely on the internal interface of the tested application:

- **Documentation.** External APIs are usually better documented than internal interfaces. This has two advantages. First, better documentation eases the implementation of the component that extracts the application state. Second, using a documented interface simplifies bug reporting and developer interaction because the state information can be provided in a format that is familiar to the developers.
- **Stability.** External APIs are significantly more stable than internal interfaces of the applications because APIs constitute a (long-term) implicit contract with users, who over time develop dependencies on it. As a consequence, the burden of updating the state extraction implementation, over the development lifetime of the application, is reduced.
- **Portability.** External APIs are frequently shared by different applications. For example, SQL is a popular interface and is nearly ubiquitously supported by relational database management systems. Relying exclusively on APIs allows the implementations of the state extraction mechanism to be reused among the various applications that support the respective APIs.

8.2 Exposing concurrency bugs due to weak-memory models

Many widely used multiprocessor architectures, such as the x86 architecture, do not enforce a global order of instructions for performance reasons. Instead, these processors implement memory models that are weaker than sequential consistency. As a consequence, there is a class of concurrency bugs that arises from these weaker memory models, which can not be exposed with our implementation of SKI, given that SKI currently implements a strong memory model. This type of concurrency bugs is particularly challenging, both for developers and for testing tools, because these bugs depend on the subtle details of the processor architecture, which are not always well specified and are often hard to reason about.

Concurrency bugs of this type can easily be introduced when shared memory accesses are not synchronized. Therefore, software that does not comply with the best-practices of concurrent programming [Adv10], which strongly discourages the deliberate use of unsynchronized shared memory accesses, is particularly susceptible to this class of concurrency bugs. In practice, developers avoid synchronization because it incurs in significant performance costs and try, instead, to develop alternatives with better performance that they believe are correct – especially with regard to performance-critical components. Researchers have shown that, unfortunately, such ad-hoc solutions are often incorrect [XPZ+10].

When testing the kernel to evaluate SKI, we noticed that kernel developers rely extensively on data races, which developers expect to be benign. This observation leads us to believe that kernels may be significantly vulnerable to this type of concurrency bugs and that developing effective testing techniques for this class of concurrency bugs would constitute an important contribution to the robustness of kernels.

8.3 Exploring the input space

In addition to detecting bugs and exploring the interleaving space, dynamic testing also requires exploring the input space. As discussed in Section 7.4, exploring the input space is, in general, a hard problem that becomes harder in the context of concurrency bugs, given that multiple threads are expected to receive input.

In this dissertation, for evaluating our tools, we conducted the exploration of the input space using naive techniques, which rely on adapting existing test cases. However, developing effective techniques to explore the input space is an orthogonal problem that could significantly further increase the effectiveness of testing. In this section we briefly discuss two approaches to explore the input space with regard to concurrent testing.

8.3.1 Profiling-based generation of tests

An approach that we believe to be promising consists in exploring the possibility of leveraging systematic tools, like SKI, to also effectively explore the input space of the tested software. SKI achieves a very high degree of control over the executed tests and this is an important property that may enable the discovery of clever algorithms to explore the input space. More specifically, the fact that SKI tests always start from the exact same initial state, leveraging the snapshot mechanism, combined with the fact that SKI tests are nearly deterministic[2] could open an opportunity for profiling-based techniques.

In the context of testing kernels, one possible profiling-based approach consists of leveraging an existing non-concurrent test suite (composed of an array of tests that issue system calls by a single thread) and profiling each test when executed by a single CPU from a certain initial state. This profiling phase would have linear complexity – each test would be profiled once for a given initial state – and serve to gather information about the instructions that are executed and the memory that is accessed. Subsequently, according to suitable algorithms, yet to be developed, pairs of tests would be selected to execute concurrently under SKI to expose concurrency bugs.

8.3.2 Leveraging bug reports

Bug reports are important for developers to find and diagnose bugs. However, bug reports are often not complete. For example, our experience reproducing concurrency bugs revealed that it is typical for concurrency crash bug reports to include information about the stack trace, which indirectly informs developers about one of the system calls invoked, but provides little or no information regarding the other concurrent system call.

In the context of diagnosing reported bugs, we envision a testing infrastructure that automatically leverages the information contained in the bug reports to select tests. For example, the testing infrastructure could automatically parse bug reports to identify stack traces and generate test cases that could reproduce the respective bugs. Additionally, it might be possible to automatically leverage other information from bug reports, either by using natural language processing techniques or by analyzing fields from the bug report system (e.g., Bugzilla) that may be relevant to reproduce the bug (e.g., software version or identification of the component).

[2]Even though currently SKI does not ensure deterministic hardware input, the analysis of experimental results showed that, in practice, this aspect of our implementation has limited impact on the execution of the tests. Furthermore, support for deterministic hardware input can be implemented if deemed necessary to support additional testing techniques.

9 Conclusions

This work improves our understanding of real-world concurrency bugs by presenting a study on their effects. Our study on concurrency bugs analyzed MySQL, a particularly complex and critical application, and provided concrete evidence of the existence of two classes of particularly difficult to detect bugs: semantic concurrency bugs and latent concurrency bugs. Taking into consideration these findings, this dissertation proposes a novel approach that detects both of these classes of concurrency bugs. The key idea behind our approach is to use linearizability as a way to infer the specification of the application. We show that this approach is applicable to complex applications, namely MySQL, and we describe effective methods to compare the state of different executions and to address false positives.

In addition, this dissertation extends to kernel-code the general approach of systematic testing for concurrency bugs. We propose a virtual machine based systematic approach that achieves fine-grained control over the kernel thread interleavings and, according to a generalized scheduling algorithm that we propose, effectively explores the kernel interleaving space. In addition, we developed several critical optimizations to ensure efficiency and we validated our approach by reproducing known concurrency bugs and by finding unknown concurrency bugs in large-scale commodity kernels, namely Linux and FreeBSD.

Despite the contributions of this dissertation, it is clear that ensuring the reliability of software, specially with regard to concurrent and complex software, will remain a hard problem. Therefore, we believe that software developers, and users, will greatly benefit from additional improvements within this field of research.

Bibliography

[ABD$^+$09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[AC77] A. Avizienis and Liming Che. On the implementation of n-version programming for software fault tolerance during execution. In *In Proc. of IEEE COMPSAC77*, 1977.

[ADADB$^+$06] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-smart disk systems: Past, present, and future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, March 2006.

[Adv10] Sarita Adve. Data races are evil with no exceptions: Technical perspective. *Commun. ACM*, 53(11):84–84, November 2010.

[AFI$^+$08] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of POWER and ARM multiprocessor machine code. In *Proc. of Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2008.

[AH09] John Admanski and Steve Howard. Autotest – Testing the untestable. In *Proc. of Ottawa Linux Symposium (OLS)*, 2009.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[Ait02] Dave Aitel. The advantages of block-based protocol analysis for security testing. Technical report, Immunity, Inc., 2002.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[Ant14] Gary Anthes. Researchers simplify parallel programming. *Commun. ACM*, 57(11):13–15, October 2014.

[API] MultiProcessor Specification. http://www.intel.com/design/archives/processors/pro/docs/242016.htm.

[AWHF10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proc. of Operating System Design and Implementation (OSDI)*, 2010.

[BA] Bug 14416 - Null pointer dereference in fs/pipe.c . http://bugzilla.kernel.org/show_bug.cgi?id=14416.

[BA2] FS: pipe.c null pointer dereference. https://git.kernel.org/cgit/linux/kernel/git/stable/stable-queue.git/tree/queue-2.6.31/fs-pipe.c-null-pointer-dereference.patch?id=36e97dec52821f76536a25b763e320eb 7434c2a5.

[BAAS09] Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proc. of Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[BAEFU06a] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Noise makers need to know where to be silent – Producing schedules that find bugs. In *Proc. of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2006.

[BAEFU06b] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In *Proc. of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2006.

[BB] Bug 22602 - Oops while unmounting an USB key with a FAT filesystem. https://bugzilla.kernel.org/show_bug.cgi?id=22602.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proc. of European Conference on Computer Systems (EuroSys)*, 2006.

[BC] Patch "ext4: fix crash when accessing /proc/mounts concurrently" has been added to the 3.6-stable tree. http://www.mail-archive.com/stable@vger.kernel.org/msg19380.html.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[BCM10] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional detection of data races. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2010.

[BD] Bug 144061 - [socket] race on unix socket close. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=144061.

[BDMT10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. *SIGPLAN Not.*, 45(6):330–340, 2010.

[BE04] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. of International Conference on Software Engineering (ICSE)*, 2004.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of Annual Technical Conference (ATC)*, 2005.

[BFM$^+$05] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[BH07] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40:33–37, December 2007.

[BKMN10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[BLA] GE Energy acknowledges blackout bug. http://www.securityfocus.com/news/8032.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[Blu12] Ben Blum. *Landslide: Systematic dynamic race detection in kernel space*. MS thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 2012. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-12-118.pdf.

[BM83] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, September 1983.

[BMP+06] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. *Lecture Notes in Computer Science*, 4168:684–695, 2006.

[BP88] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Softw. Eng.*, 14(10):1462–1477, October 1988.

[BP05] Daniel P. Bovet and Marco Cesati Ph. *Understanding the Linux Kernel*. O'Reilly Media, third edition edition, 2005.

[BPSZ10] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 301–310, New York, NY, USA, 2010. ACM.

[BS09] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *Proc. of European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009.

[BSADAD09] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating file-system mistakes with EnvyFS. In *Proc. of Annual Technical Conference (ATC)*, 2009.

[BYLN09] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[CBM10] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[CC00] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2000.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of Operating System Design and Implementation (OSDI)*, 2008.

[CF04] George Candea and Armando Fox. End-user effects of microreboots in three-tiered internet systems. *ArXiv Computer Science e-prints*, March 2004.

[CFS] Linux: The Completely Fair Scheduler. http://kerneltrap.org/node/8059.

[CGP$^+$08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):1–38, 2008.

[CHWL09] Chun-Ting Chen, Chun-Chen Hsu, Jan-Jan Wu, and Pangfeng Liu. GFS: A distributed file system with multi-source data access and replication for grid computing. In *Proc. of International Conference on Advances in Grid and Pervasive Computing (GPC)*, 2009.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[CKF$^+$04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A technique for cheap recovery. In *Proc. of Operating System Design and Implementation (OSDI)*, 2004.

[CKZ13a] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proc. of European Conference on Computer Systems (EuroSys)*, 2013.

[CKZ$^+$13b] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2013.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proc. of Operating System Design and Implementation (OSDI)*, 1999.

[Cla76] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[CLC13] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *Proc. of International Conference on Virtual Execution Environments (VEE)*, 2013.

[CLL$^+$02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2002.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proc. of Hot Topics in Operating Systems (HotOS)*, 2001.

[Cor] Oracle Corporation. MySQL Bugs: How to Report a Bug. `https://bugs.mysql.com/how-to-report.php`.

[COV] Software testing and static analysis tools — coverity. `http://www.coverity.com/`.

[CRCM12] João Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *Proc. of European Conference on Computer Systems (EuroSys)*, 2012.

[CRL03] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21:236–269, August 2003.

[CSL$^+$13] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2013.

[CWG$^+$11] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2011.

[CWTY10] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proc. of Operating System Design and Implementation (OSDI)*, 2010.

[CYC$^+$01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2001.

[DHH09] Daniel Dawson, Nathan Hawes, and Christian Hoermann. Finding bugs in open source kernels using Parfait. In *Proc. of Kernel Conference Australia (KCA)*, 2009.

[DLCO09] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[DLFC08] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proc. of International Conference on Virtual Execution Environments (VEE)*, 2008.

[DZW$^+$14] Zhuofang Dai, Zheng Zhang, Haojun Wang, Yi Li, and Weihua Zhang. Parallelized race detection based on gpu architecture. In Junjie Wu, Haibo Chen, and Xingwei Wang, editors, *Advanced Computer Architecture*, volume 451 of *Communications in Computer and Information Science*, pages 113–127. Springer Berlin Heidelberg, 2014.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *SIGOPS Operating Systems Review*, 37(5):237–252, 2003.

[EBA+13] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[EBNS13] Tayfun Elmas, Jacob Burnim, George C. Necula, and Koushik Sen. CONCURRIT: A domain specific language for concurrency bugs. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2013.

[EFG+03] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency - Practice and Experience*, 15(3-5):485–499, 2003.

[EMBO10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proc. of Operating System Design and Implementation (OSDI)*, 2010.

[EQT07] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2007.

[EU04] Yaniv Eytani and Shmuel Ur. Compiling a benchmark of documented multi-threaded bugs. *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, 17:266a, 2004.

[FBG12] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of Symposium on Principles of Programming Languages (POPL)*, 2004.

[FF09] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, 2009.

[FLR11] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proc. of European Conference on Computer Systems (EuroSys)*, 2011.

[FLSR10] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2010.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[FRB14] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proc. of Operating System Design and Implementation (OSDI)*, 2014.

[FSS] Fsstress source code. http://cvs.sourceforge.net/viewcvs.py/ltp/ltp/testcases/kernel/fs/fsstress/.

[FZ10] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[GEE] GE Energy acknowledges blackout bug. http://www.securityfocus.com/news/8032.

[GHK$^+$01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001.

[GKRY03] Weining Gu, Zbigniew Kalbarczyk, Iyer K. Ravishankar, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *Proc. of International Conference on Dependable Systems and Networks (DSN)*, 2003.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of Symposium on Principles of Programming Languages (POPL)*, 1997.

[Gol08] Gene Golovchinsky. Reading in the office. In *Proceedings of the 2008 ACM Workshop on Research Advances in Large Digital Book Repositories*, BooksOnline '08, pages 21–24, New York, NY, USA, 2008. ACM.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Proc. of Reliability in Distributed Software and Database Systems (SRDS)*, 1986.

[GZNM11] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 395–404, New York, NY, USA, 2011. ACM.

[GZTQ09] Qi Gao, Wenbin Zhang, Yan Tang, and Feng Qin. First-aid: Surviving and preventing memory management bugs during production runs. In *Proc. of European Conference on Computer Systems (EuroSys)*, 2009.

[HC14] Jennia Hizver and Tzi-cker Chiueh. Real-time deep virtual machine introspection and its applications. In *Proc. of International Conference on Virtual Execution Environments (VEE)*, 2014.

[HHS13] Ruirui Huang, Erik Halberg, and G. Edward Suh. Non-race concurrency bug detection through order-sensitive critical sections. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 2013.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[Hol97] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[Hol14] Gerard J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, 2014.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[HYN+14] 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 *Proc. of Programming Languages Design and Implementation (PLDI)*, 2014.

[IEE94] IEEE. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System Application Program Interface (API), Amendment 1: Realtime Extension (C Language), IEEE Std 1003.1b-1993*. IEEE Standards Office, New York, NY, USA, 1994.

[IEE95] IEEE. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX): System Application Program Interface (API), Amendment 2: Thread Extensions, IEEE Std 1003.1c-1995*. IEEE Standards Office, New York, NY, USA, 1995.

[IEE08] IEEE. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Base Specifications, IEEE Std 1003.1-2008*. IEEE Standards Office, New York, NY, USA, 2008.

[INTa] Intel's billion-dollar mistake: Why chip flaws are so hard to fix. http://venturebeat.com/2011/01/31/intels-billion-dollar-mistake-why-chip-flaws-are-so-hard-to-fix/.

[INTb] Intel Previews Intel Xeon 'Nehalem-EX' Processor. http://www.intel.com/pressroom/archive/releases/2009/20090526comp.htm.

[JADAD06] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of Annual Technical Conference (ATC)*, 2006.

[JSS+12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2012.

[JTZC08] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proc. of Operating System Design and Implementation (OSDI)*, 2008.

[KCC10] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *Proc. of Annual Technical Conference (ATC)*, 2010.

[KER] Kernel threads made easy. http://lwn.net/Articles/65178/.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[Kiv07] Avi Kivity. KVM: The Linux virtual machine monitor. In *Proc. of Ottawa Linux Symposium (OLS)*, 2007.

[KK98] I. Koren and Z. Koren. Defect tolerance in vlsi circuits: techniques and yield analysis. *Proceedings of the IEEE*, 86(9):1819–1838, Sep 1998.

[KL93] Harry Koehnemann and Timothy E. Lindquist. Towards target-level testing and debugging tools for embedded software. In *Proc. of TRI-Ada*, 1993.

[KTGN10] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production MapReduce cluster. In *Proc. of International Conference on Cluster, Cloud and Grid Computing (CC-GRID)*, 2010.

[KWLM09] Terence Kelly, Yin Wang, Stephane Lafortune, and Scott Mahlke. Eliminating concurrency bugs with control engineering. *IEEE Computer*, 99(1), 2009.

[KZC12] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with Portend. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[KZC13] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2013.

[LADADL13] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *Proc. of Conference on File and Storage Technologies (FAST)*, 2013.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[Lar02] Paul Larson. Testing linux with linux test project. In *Proc. of Ottawa Linux Symposium (OLS)*, 2002.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2011.

[LCS10] Brandon Lucia, Luis Ceze, and Karin Strauss. Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 2010.

[Liu07] Xuezheng Liu. WiDS checker: Combating bugs in distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)*, 2007.

[LLKB12] Di Liu, Matthew Lease, Rebecca Kuipers, and Randolph G. Bias. Crowdsourcing for usability testing. *CoRR*, 2012.

[LLS$^+$13] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? Findings from a Google case study. In *Proc. of International Conference on Software Engineering (ICSE)*, 2013.

[LOCa] Nasty Lockup Issue Still Being Investigated For Linux 3.18. http://www.phoronix.com/scan.php?page=news_item&px=MTg1MDc.

[LOCb] ANNOUNCE: Lock validator. http://lwn.net/Articles/185605/.

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGARCH Computer Architecture News*, 36(1):329–339, 2008.

[LT93] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[LTQZ06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[LTW+06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open-source software. In *Proc. of Architectural and System Support for Improving Software Dependability (ASID)*, 2006.

[LVN10] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2010.

[LVT+11] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2011.

[LZL+13a] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proc. of International Conference on Software Engineering (ICSE)*, 2013.

[LZL+13b] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proc. of International Conference on Software Engineering (ICSE)*, 2013.

[LZL+13c] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proc. of International Conference on Software Engineering (ICSE)*, 2013.

[McC02] Dave McCracken. POSIX threads and the Linux kernel. In *Proc. of Ottawa Linux Symposium (OLS)*, 2002.

[MDADAD13] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Ffsck: The fast file system checker. In *Proc. of Conference on File and Storage Technologies (FAST)*, 2013.

[Mic] Microsoft. Generating test data for databases by using data generators. http://msdn.microsoft.com/en-us/library/dd193262.aspx.

[MKZ08] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *Proc. of International Conference on Management of Data (SIGMOD)*, pages 499–510, 2008.

[MMN09] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2009.

[MQB$^+$08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of Operating System Design and Implementation (OSDI)*, 2008.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proc. of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 509–518, "Las Vegas, NV", October 1998.

[MW71] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, March 1971.

[MYI] The MyISAM storage engine. http://dev.mysql.com/doc/refman/5.0/en/myisam-storage-engine.html.

[MYSa] MySQL :: The world's most popular open-source database. http://www.mysql.com.

[MYSb] MySQL Bugs. http://bugs.mysql.com.

[MYSc] MySQL :: Market Share. http://www.mysql.com/why-mysql/marketshare/.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2006.

[NB05] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proc. of International Conference on Software Engineering (ICSE)*, 2005.

[NBH08] Kara Nance, Matt Bishop, and Brian Hay. Virtual machine introspection: Observation or interference? *IEEE Security and Privacy*, 6(5):32–37, September 2008.

[NBMM12a] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2012.

[NBMM12b] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2012.

[NJT13] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Proc. of Working Conference on Mining Software Repositories (MSR)*, 2013.

[NPCF08] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[NPSG09] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proc. of International Conference on Software Engineering (ICSE)*, 2009.

[NVI] GeForce GTX 295. http://www.nvidia.com/object/product_geforce_gtx_295_us.html.

[NWT$^+$07] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2007.

[NZHZ07] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proc. of Conference on Computer and Communications Security (CCS)*, 2007.

[OPE] OpenSUSE News. https://news.opensuse.org/2014/03/19/development-for-13-2-kicks-off/.

[OWB05] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering (TSE)*, 31(4):340–355, April 2005.

[Pac07] Sasha Pachev. *Understanding MySQL internals*. O'Reilly Media, Inc., 2007.

[PFMA04] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - A coprocessor-based kernel runtime integrity monitor. In *Proc. of Conference on USENIX Security Symposium (SSYM)*, 2004.

[PLM$^+$10a] Wei-Feng Pan, Bing Li, Yu-Tao Ma, Ye-Yi Qin, and Xiao-Yan Zhou. Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. *Journal of Computer Science and Technology*, 25(6):1202–1213, 2010.

[PLM$^+$10b] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of I/O workload in virtualized cloud environments. In *Proc. of International Conference on Cloud Computing (CLOUD)*, 2010.

[PLZ09] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[PSST08] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. CPU demand for web serving: Measurement analysis and dynamic estimation. *Perform. Eval.*, 65:531–553, June 2008.

[PVSD12] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2012.

[PZX$^+$09] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2009.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—A safe method to survive software failures. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2005.

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proc. of European Conference on Computer Systems (EuroSys)*, 2009.

[RKBD14] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proc. of International Conference on Software Engineering (ICSE)*, 2014.

[RKS12] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing drivers without devices. In *Proc. of Operating System Design and Implementation (OSDI)*, 2012.

[RKW$^+$06] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)*, 2006.

[RLT78] Brian Randell, Peter Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, 1978.

[RSSK14] Ajitha Rajan, Subodh Sharma, Peter Schrammel, and Daniel Kroening. Accelerated test execution using gpus. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 97–102, New York, NY, USA, 2014. ACM.

[Rus] Paul Rusty Russell. Unreliable Guide To Locking. http://kernelbook.sourceforge.net/kernel-locking.pdf.

[RVS13] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, April 1985.

[Ryz10] Leonid Ryzhyk. *On the Construction of Reliable Device Drivers.* PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Jan 2010.

[SBN$^+$97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, 1997.

[SC92] Mark Sullivan and Ram Chillarege. A comparison of software defects in database management systems and operating systems. In *Proc. of International Symposium on Fault-Tolerant Computing (FTCS)*, 1992.

[SC07] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *Tests and Proofs*, volume 4454 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2007.

[SCA09] Swarup K Sahoo, John Criswell, and Vikram S. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. Tech. Report 2142/13697, University of Illinois, University of Illinois, 2009.

[SCA10] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proc. of International Conference on Software Engineering (ICSE)*, 2010.

[Sch95] Beth A. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, 28(6):72–78, Jun 1995.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2008.

[SKAZ04] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proc. of Annual Technical Conference (ATC)*, 2004.

[SQL] SQL Standards. http://www.jcc.com/resources/sql-standards.

[SSN+09] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. of Symposium on Principles of Programming Languages (POPL)*, 2009.

[SSV13] Jo M. Silva, Jos Simo, and Lus Veiga. Ditto deterministic execution replayability-as-a-service for java vm on multiprocessors. In David Eyers and Karsten Schwan, editors, *Middleware 2013*, volume 8275 of *Lecture Notes in Computer Science*, pages 405–424. Springer Berlin Heidelberg, 2013.

[STO] Storage engine poll. http://dev.mysql.com/doc/refman/5.0/en/storage-engines.html.

[Sto02] Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. of Workshop on Runtime Verification (RV)*, 2002.

[SWS+09] Martin Süßkraut, Stefan Weigert, Ute Schiffel, Thomas Knauth, Martin Nowack, Diogo Becker de Brum, and Christof Fetzer. Speculation for parallelizing runtime checks. In *Proc. of International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2009.

[TER] Tera-BLAST White Paper - TimeLogic. http://www.timelogic.com/documents/TimeLogic_Tera-BLAST_whitepaper_v1.0.pdf.

[TLL+14] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open-source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[TRI] Trinity: A Linux system call fuzzer. http://codemonkey.org.uk/projects/trinity/.

[TUYT07] Rachel Tzoref, Shmuel Ur, and Elad Yom-Tov. Instrumenting where it hurts: An automatic concurrent debugging technique. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[UBH09] Iñigo Urteaga, Kevin Barnhart, and Qi Han. REDFLAG: A run-time, distributed, flexible, lightweight, and generic fault detection service for data-driven wireless sensor applications. *Pervasive Mob. Comput.*, 5:432–446, October 2009.

[ULSD04] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of Conference on Virtual Machine Research And Technology Symposium (VM)*, 2004.

[uTe] uTest. White paper: Crowdsourced usability testing. http://alexcrockett.com/wp-content/uploads/downloads/Books/Crowdsourced_Usability_Testing.pdf.

[Vaf10] Viktor Vafeiadis. Automatically proving linearizability. In *Proc. of International Conference on Computer Aided Verification (CAV)*, 2010.

[Val94] John D. Valois. Implementing lock-free queues. In *Proc. of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 1994.

[VBLM07] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2007.

[VCFN11] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2011.

[VEG] Azul Systems - Industry's Leading Azul Compute Appliances. http://www.azulsystems.com/products/compute_appliance.htm.

[VFS14] Dhaval Vyas, Thomas Fritz, and David Shepherd. Bug reproduction : a collaborative practice within software maintenance activities. In C. Rossitto, Luigina Ciolfi, David Martin, and B. Conein, editors, *COOP 2014 - Proceedings of the 11th International Conference on the Design of Cooperative Systems*, IFIP – The International Federation for Information Processing. Spriger, Nice, France, May 2014.

[Viz07] Mike Vizard. The yin and yang of software development. *Queue*, 5(4), May 2007.

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Proc. of European Software En-*

gineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE), 2007.

[VKM02] Udaykiran Vallamsetty, Krishna Kant, and Prasant Mohapatra. Characterization of e-commerce traffic. In *Proc. of Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, 2002.

[VT06] Lucian Voinea and Alexandru Telea. How do changes in buggy Mozilla files propagate? In *Proc. of Symposium on Software Visualization (SoftVis)*, 2006.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *Proc. of SPIN Workshop on Model Checking Software (SPIN)*, 2009.

[VZ91] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 25(5):26–40, September 1991.

[WJKT05] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. of International Conference on Testing of Communicating Systems (Testcom)*, 2005.

[WLC$^+$11] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: A scalable and portable parallel full-system emulator. In *Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[WS06] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, February 2006.

[WTH$^+$12] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and precise analysis of parallel programs through schedule specialization. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2012.

[XBH05a] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[XBH05b] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proc. of Programming Languages Design and Implementation (PLDI)*, 2005.

[XMS+07] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. of Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

[XPZ+10] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proc. of Operating System Design and Implementation (OSDI)*, 2010.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)*, 2009.

[YCW+14] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 2014.

[YKKK09] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proc. of Networked Systems Design and Implementation (NSDI)*, 2009.

[YMZ+11] 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 *Proc. of Symposium on Operating System Principles (SOSP)*, 2011.

[YNPP12] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2005.

[YSE06] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proc. of Operating System Design and Implementation (OSDI)*, 2006.

[YTEM04] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proc. of Operating System Design and Implementation (OSDI)*, 2004.

[ZACS11] Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. Debug determinism: The sweet spot for replay-based debugging. In *Proc. of Hot Topics in Operating Systems (HotOS)*, 2011.

[ZAH11] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: A case study on Firefox. In *Proc. of Working Conference on Mining Software Repositories (MSR)*, 2011.

[Zim95] Philip Zimmermann. *PGP Source Code and Internals*. MIT Press, Cambridge, MA, USA, 1995.