Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics

Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou

Department of Computer Science,
University of Illinois at Urbana Champaign, Urbana, IL 61801
{shanlu,soyeon,eseo2,yyzhou}@uiuc.edu

Abstract

The reality of multi-core hardware has made concurrent programs pervasive. Unfortunately, writing correct concurrent programs is difficult. Addressing this challenge requires advances in multiple directions, including concurrency bug detection, concurrent program testing, concurrent programming model design, etc. Designing effective techniques in all these directions will significantly benefit from a deep understanding of real world concurrency bug characteristics.

This paper provides the first (to the best of our knowledge) comprehensive real world concurrency bug characteristic study. Specifically, we have carefully examined concurrency bug patterns, manifestation, and fix strategies of 105 randomly selected real world concurrency bugs from 4 representative server and client open-source applications (MySQL, Apache, Mozilla and OpenOffice). Our study reveals several interesting findings and provides useful guidance for concurrency bug detection, testing, and concurrent programming language design.

Some of our findings are as follows: (1) Around one third of the examined non-deadlock concurrency bugs are caused by violation to programmers' order intentions, which may not be easily expressed via synchronization primitives like locks and transactional memories; (2) Around 34% of the examined non-deadlock concurrency bugs involve multiple variables, which are not well addressed by existing bug detection tools; (3) About 92% of the examined concurrency bugs can be reliably triggered by enforcing certain orders among no more than 4 memory accesses. This indicates that testing concurrent programs can target at exploring possible orders among every small groups of memory accesses, instead of among all memory accesses; (4) About 73% of the examined non-deadlock concurrency bugs were not fixed by simply adding or changing locks, and many of the fixes were not correct at the first try, indicating the difficulty of reasoning concurrent execution by programmers.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; B.8.1 [*Hardware*]: Performance and Reliability–Reliability, Testing, and Fault-Tolerance

General Terms Languages, Reliability

Keywords concurrent program, concurrency bug, bug characteristics

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

1. Introduction

1.1 Motivation

Concurrent programs are becoming prevalent due to the reality of multi-core hardware. Nowadays, not only high-end servers but also desktop machines need concurrent programs to make the best use of their multi-core hardware. As a result, the difficulty of concurrent programming is hitting the entire software development community, rather than just the elite few. Writing good quality concurrent programs has become critically important.

Unfortunately, writing correct concurrent programs is difficult. Most programmers think sequentially and therefore they make mistakes easily when writing concurrent programs. Even worse, the notorious non-determinism of concurrent programs makes concurrency bugs difficult to repeat during interactive diagnosis.

Addressing the above challenges will require efforts from multiple related directions including those listed as follows, all of which have made some progress over the past years but still have many open, unsolved issues:

(1) Concurrency bug detection Most previous concurrency bug detection research has focused on detecting data race bugs [7, 10, 31, 33, 37, 42] and deadlock bugs [3, 10, 37]. Data race occurs when two conflicting accesses to one shared variable are executed without proper synchronization, e.g., not protected by a common lock. Deadlock occurs when two or more operations circularly wait for each other to release the acquired resource (e.g., locks). Recently, several approaches have also been proposed to detect atomicity-violation bugs [12, 23, 41], which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region.

Although previous work has proposed effective methods to detect certain types of concurrency bugs, it is still far from providing a complete solution. In particular, several open questions about concurrency bug detection still remain: (i) Can existing bug detection tools *detect all* real world concurrency bugs? Specifically, what types of concurrency bugs exist in real world? Is there any type that has not been addressed yet by existing work? In addition, are the assumptions of existing tools about concurrency bugs valid? For example, most previous race detection and many atomicity bug detection techniques focus on synchronization among accesses to a single variable. How many concurrency bugs are missed by this single variable assumption? (ii) How helpful are existing tools in *diagnosing and fixing* the real world concurrency bugs detected by them? For example, many concurrency bug detection tools remind programmers that some conflicting accesses are not protected by the same lock. Such information can help programmers add or change lock operations. However, how often are real world bugs fixed by adding or changing lock operations? More generally, how do pro-

grammers fix real world concurrency bugs and what information do they need?

(2) Concurrent program testing and model checking Testing is a common practice in software development. It is a critical step for *exposing* software bugs before release.

Existing testing techniques mainly focus on the *sequential* aspects of programs, such as statements, branches, etc. and can not effectively address concurrent programs' *concurrency* aspects, such as multi-thread (or multi-process) interleavings [28].

The major challenge of concurrency testing is the exponential *interleaving* space of concurrent programs. Exposing concurrency bugs requires not only a bug-exposing input, but also a bug-triggering execution interleaving. Therefore, to achieve a complete testing coverage of concurrent programs, testing needs to cover every possible interleaving for each input test case [39], which is infeasible in practice.

To address the above challenge, an open question in concurrency testing is as follows: can we selectively test a small number of representative interleavings and still expose most of the concurrency bugs? Motivated by this problem, previous work such as the ConTest project [4, 9] has proposed some methods to perturb program execution and force certain interleavings by injecting artificial delays after every synchronization point. While an inspiring attempt, it is unclear, both quantitatively and qualitatively, what portion of concurrency bugs can be exposed by such heuristics.

Ultimately, designing practical and effective test cases for concurrent programs requires a good understanding of *the manifestation conditions of real world concurrency bugs*. That is, we need to know what conditions are needed, besides program inputs, to reliably trigger a concurrency bug. Specifically, how many threads, how many variables, and how many accesses are usually involved in a real world concurrency bug's manifestation?

Similar questions are also encountered in software verification and model checking [13, 28, 34] for concurrent programs. Better understanding of the manifestation of real world concurrency bugs can help model checking prioritize the program states and alleviate its state explosion problem.

(3) Concurrent programming language design Good concurrent programming languages can help programmers correctly express their intentions and therefore avoid certain types of concurrency bugs. Along this direction, transactional memory (TM) [1,2,15,16,25,26,27,36] is one of the popular trends. TM provides programmers an easier way to specify which code regions should be atomic. Further, it automatically protects the atomicity of the specified region against other specified regions through underlying hardware and software support.

Although TM shows great potential, there are many open questions, including (i) What portion of bugs can be avoided by using TM? (ii) What are the real world concerns that TM design needs to pay attention to? (iii) Besides TM, what other programming language supports will be useful for programmers to write correct concurrent programs?

Addressing the open questions in all of the above directions will significantly benefit from a better understanding of real world concurrency bug characteristics—basically, we can learn from the common mistakes programmers are making in writing concurrent programs. For example, if many real world concurrency bugs involve multiple shared variables, we need to extend concurrency bug detection techniques to address multi-variable concurrency bugs; if the manifestation of most real world concurrency bugs are guaranteed by a partial order among only two threads, concurrent program testing only needs to cover pairwise interleavings for every pair of program threads; if there are some concerns in avoiding real world concurrency bugs with existing synchronization primitives, we can extend transactional memory model or design new language support to further ease writing concurrent programs; if a certain type of information is frequently used by programmers in fixing real world concurrency bugs, bug detection tools can be extended to provide such information and thus become more useful in practice.

In the past, many empirical studies on general program bug characteristics (not specific to concurrency bugs) have been done. Their findings have provided useful guidelines and motivations for bug detection, testing and programming language design. For example, the study of bug types in IBM software systems [38] in 1990's demonstrated the importance of memory bugs and has motivated many commercial and open-source memory bug detection tools such as Purify [18], Valgrind [30], CCured [29], etc. A recent study of operating system bugs [8] revealed that copy-paste was an important cause of semantic bugs, and has inspired a tool called CP-Miner that focused on detecting copy-pasted code and semantic bugs related to copy-paste [19].

Unfortunately, few studies have been conducted on real world concurrency bug characteristics. Previously, researchers realizing the importance of such a study have conducted a preliminary work on concurrency bug characteristics [11]. However, they built their observations upon programs that were intentionally made buggy by students for the characteristic study.

The lack of a good real-world concurrency bug characteristic study is mainly due to the following two reasons:

(1) It is difficult to collect real world concurrency bugs, especially since they are usually under-reported. As observed in previous work [6], the non-determinism hindered the users from reporting concurrency bugs, and made concurrency bug reports difficult to get understood and solved by programmers. Therefore, it is time-consuming to collect a good set of real world concurrency bugs.

(2) Concurrency bugs are not easy to understand. Their patterns and manifestations usually involve complicated interactions among multiple program components, and are therefore hard to understand.

1.2 Contributions

This work provides the first (to the best of our knowledge) comprehensive real world concurrency bug characteristic study. Specifically, we examine the *bug patterns, manifestations, fix strategies* and other characteristics of real world concurrency bugs. Our study is based on *105* randomly selected real world concurrency bugs, including *74* non-deadlock bugs and *31* deadlock bugs, collected from *4* large and mature open-source applications: MySQL, Apache, Mozilla and OpenOffice, representing both server and client applications. For each bug, we carefully examine its bug report, corresponding source code, related patches, and programmers' discussion, all of which together provide us a relatively thorough understanding of the bug patterns, manifestation conditions, fix strategies and diagnosis processes.

Our study reveals many interesting findings, which provide useful guidelines for concurrency bug detection, concurrent program testing, and concurrent programming language design. We summarize our main findings and their implications in Table 1.

While we believe that the applications and bugs we examined well represent a large body of concurrent applications, we do not intend to draw any general conclusions about all concurrent applications. In particular, we should note that all of the characteristics and findings obtained in this study are associated with the four examined applications and the programming languages these applications use. Therefore, the results should be taken with the specific applications and our evaluation methodology in mind (see Section 2.3 for our discussion about threats to validity).

Findings on Bug Patterns (Section 3)	**Implications**
(1) Almost all (97%) of the examined non-deadlock bugs belong to one of the *two simple bug patterns*: atomicity-violation or order-violation*.	Concurrency bug detection can focus on these two bug patterns to detect most concurrency bugs.
(2) About one third (32%) of the examined non-deadlock bugs are *order-violation bugs*, which are *not* well addressed in previous work.	*New* concurrency bug detection tools are needed to detect order-violation bugs, which are not addressed by existing atomicity violation or race detectors.
Findings on Manifestation (Section 4)	**Implications**
(3) Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between *2 threads* is enforced.	Pairwise testing on concurrent program threads can expose most concurrency bugs, and greatly reduce the testing complexity.
(4) Some (22%) of the examined deadlock bugs are caused by *one thread* acquiring resource held by itself.	Single-thread based deadlock detection and testing techniques can help eliminate these simple deadlock bugs.
(5) Many (66%) of the examined non-deadlock concurrency bugs' manifestation involves concurrent accesses to *only one variable.*	Focusing on concurrent accesses to one variable is a good simplification for concurrency bug detection, which is used by many existing bug detectors.
(6) One third (34%) of the examined non-deadlock concurrency bugs' manifestation involves concurrent accesses to *multiple variables.*	*New* detection tools are needed to address *multiple variable concurrency bugs.*
(7) Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most *two resources*.	Pairwise testing on the acquisition/release sequences to two resources can expose most deadlock concurrency bugs, and reduce testing complexity.
(8) Almost all (92%) of the examined concurrency bugs are guaranteed to manifest if certain partial order among *no more than 4 memory accesses* is enforced.	Testing partial orders *among every small group* of accesses can expose most concurrency bugs, and simplify the interleaving space *from exponential to polynomial*.
Findings on Bug Fix Strategies (Section 5)	**Implications**
(9) Three quarters (73%) of the examined non-deadlock bugs are fixed by techniques *other than* adding/changing locks. Programmers need to consider correctness, performance and other issues to decide the most appropriate fix strategy.	Bug detection and diagnosis tools need to provide more bug pattern and manifestation information, besides lock information, to help programmers fix bugs.
(10) Many (61%) of the examined deadlock bugs are fixed by preventing one thread from acquiring one resource (e.g. lock). Such fix can introduce non-deadlock concurrency bugs.	Fixing deadlock bugs might introduce non-deadlock concurrency bugs. Special help is needed to ensure the correctness of deadlock bug fixes.
Findings on Bug Avoidance (Section 5.3)	**Implications**
(11) Transactional memory (TM) can help avoid about one third (39%) of the examined concurrency bugs.	Transactional memory (TM) is a promising language feature for programmers.
(12) TM *could* help avoid over one third (42%) of the examined concurrency bugs, if some *concerns* are addressed.	TM designers may need to pay attention to some concerns, such as how to protect hard-to-rollback operations.
(13) Some (19%) of the examined concurrency bugs *cannot* benefit from basic TM designs, because of their bug patterns.	Better programming language features to help express *"order"* semantics in C/C++ programs are desired.

Table 1. Our findings of real world concurrency bug characteristic and their implications for concurrency bug detection, concurrent program testing and concurrent programming language design. (*: All terms and categories mentioned here will be explained in Section 2.)

2. Methodology

2.1 Bug sources

Applications: We select four representative open source applications in our study: MySQL, Apache, Mozilla, and OpenOffice. These are all mature (with 9–13 years development history) large concurrent applications (with 1–4 million lines of code), with well maintained bug databases. These four applications represent different types of server applications (database and web server) and client applications (browser suite and office suite). Concurrency is used for different purposes in these applications. Server applications mostly use concurrency to handle concurrent client requests. They can have hundreds or thousands of threads running at the same time. Client and office applications mostly use concurrency to synchronize multiple GUI sessions and background working threads.

Bugs: We *randomly* collect concurrency bugs from the bug databases of these applications. Since these databases contain more than five hundred thousand bug reports, in order to effectively collect concurrency bugs from them, we used a large set of keywords related to concurrency bugs, for example, 'race(s)', 'deadlock(s)', 'synchronization(s)', 'concurrency' 'lock(s)', 'mutex(es)', 'atomic', 'compete(s)', and their variations. From the thousands of bug-reports that contain at least one keyword from the above keyword set, we *randomly* pick about five hundred bug reports with clear and detailed root cause descriptions, source codes, and bug fix information. Then, we manually check them to make sure that the bugs are really caused by programmers' wrong assumptions about concurrent execution, and finally get 105 concurrency bugs.

Application	Description	# of Bug Samples	
		Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Browser Suite	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Table 3. Our application set and bug set

Definitions Related to Bug Pattern Study			
Dimension	**Category**	**Description**	**Abbr.**
Bug Pattern*	Atomicity Violation	The desired serializability among multiple memory accesses is violated. (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution.)	Atomicity
	Order Violation	The desired order between two (groups of) memory accesses is flipped. (i.e. A should always be executed before B, but the order is not enforced during execution.)	Order
	Other	Concurrency bugs other than the atomicity violation and order violation.	Other

Definitions Related to Bug Manifestation Study		
Dimension	**Term**	**Definition**
Bug Manifestation	Manifestation Condition	A specific execution order among a smallest set (S) of memory accesses. As long as that order is enforced, no matter how, the bug is guaranteed to manifest.
	# of threads involved	The number of distinct threads that are included in S.
	# of variable involved	The number of distinct variables that are included in S.
	# of accesses involved	The number of accesses that are included in S.

Definitions Related to Bug Fix Study			
Dimension	**Category**	**Description**	**Abbr.**
Non-deadlock Fix Strategy	Condition Check	(1) While-flag; or (2) optimistic concurrency with consistency check.	COND
	Code Switch	Switch the order of certain statements in the source code.	Switch
	Design Change	Change the design of data structures or algorithms.	Design
	Lock Strategy	(1) Add/change locks; or (2) adjust the region of critical sections.	Lock
	Other	Strategies other than the above ones.	Other
Deadlock Fix Strategy	Give up resource	Not acquiring a resource (lock, etc.) for certain code region.	GiveUp
	Split Resource	Split a big resource to smaller pieces to avoid competition.	Split
	Change acquisition order	Switch the acquisition order among several resources.	AcqOrder
	Other	Strategies other than the above ones.	Other
Concerns in Transactional Memory	Very long code	A code region is too long to be put into a transaction.	Long
	Rollback Problem	Some I/O and system calls are hard to roll back.	Rollback
	Code Nature	Source code with certain design is hard to turn to transaction.	Nature

Table 2. Our characteristic categories and definitions. (*: The bug pattern category is determined by the root cause of a concurrency bug, i.e. what type of programmers' synchronization intention is violated, *regardless of possible bug fix strategies.*)

We separately study two types of concurrency bugs: *deadlock bugs* and *non-deadlock concurrency bugs*. These two types of bugs have completely different properties, and demand different detection, recovery approaches. Therefore, we separate them for the ease of investigation.

Finally, we collect 105 concurrency bugs, including 74 non-deadlock concurrency bugs and 31 deadlocks bugs. The details are shown in Table 3.

2.2 Characteristic categories

In order to provide guidance for future research on concurrent program reliability, in this work, we focus on three aspects of concurrency bug characteristics: bug pattern, manifestation, and bug fix strategy. Other characteristics, such as failure impact and bug diagnosis process, will be briefly discussed at the end.

(1) Along the *bug pattern* dimension, we classify non-deadlock concurrency bugs into three categories (atomicity-violation bugs, order-violation bugs and the other bugs) based on their root causes, i.e., what types of synchronization intentions are violated. Detailed definitions are shown in Table 2. Here we do not classify data race as a bug pattern. The reason is that, a data race may indicate a concurrency bug, but it can also be a benign race in many cases, e.g., while-flag. Furthermore, data-race free does not mean concurrency bug free [12, 23]. We do not further break deadlocks into subcategories as most of them are relatively similar and simple.

(2) For the *manifestation* characteristics, we study the required condition for each concurrency bug to manifest (denoted as *manifestation condition*, defined in Table 2), and then discuss concurrency bugs based on how many threads, how many variables (resources), and how many accesses are involved in their manifestation conditions.

(3) For the *bug fix strategy*, we study both the final patch's fix strategy and the mistakes in intermediate patches. We also evaluate how transactional memory can help avoid these bugs. All the related classification is shown in Table 2.

2.3 Threats to validity

Similar to the previous work, real world characteristic studies are all subject to a validity problem. Potential threats to the validity of our characteristic study are the representativeness of the applications, concurrency bugs used in our study, and our examination methodology.

As for application representativeness, our study chooses four server and client-based concurrent applications written in C/C++, which are the popular programming languages for these types of applications. We believe that these four applications well represent server and client-based concurrent applications, which are two large classes of concurrent applications. However, our study may not reflect the characteristics of other types of applications, such as scientific applications, operating systems, or applications written in other programming languages (e.g., Java).

As for bug representativeness, the concurrency bugs we studied are *randomly* selected from the bug database of the above applications. They provide good samples of the fixed bugs in those applications. While characteristics of non-fixed or non-reported concurrency bugs might be different, these bugs are not likely as important as the reported and fixed bugs that are examined in our study.

In terms of our examination methodology, we have examined every piece of information related to each examined bug, including programmers' clear explanations, forum discussions, source code patches, multiple versions of source codes, and bug-triggering test cases. In addition, we are also familiar with the examined applications, since we have modified and used them in many of our previously published work [22, 23, 35].

Overall, while our conclusions cannot be applied to *all* concurrent programs, we believe that our study does capture the characteristics of concurrency bugs in two large important classes of concurrent applications: server-based and client-based applications. In addition, most of these characteristics are consistent across all four examined applications, indicating the validity of our evaluation methodology to some degree. Additionally, we do not emphasize

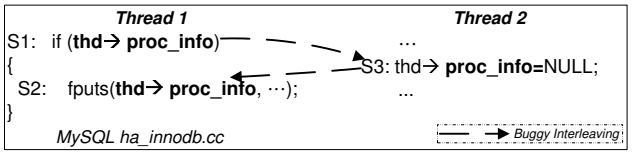

Figure 1. An atomicity violation bug from MySQL.

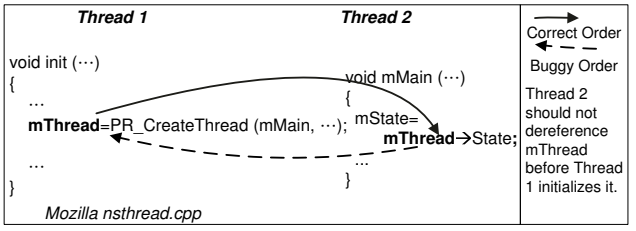

Figure 2. An order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 should not read `mThread` until thread 1 initializes `mThread`. Note that, this bug could be fixed by making `PR_CreateThread` atomic with the write to `mThread`. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

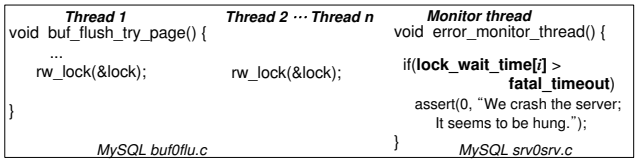

Figure 3. A MySQL bug that is neither an atomicity-violation bug nor an order-violation bug. The monitor thread is designed to detect deadlock. It restarts the server when a thread i has waited for a lock for more than `fatal_timeout` amount of time. In this bug, programmers under-estimate the workload (`n` could be very large), and therefore the lock waiting time would frequently exceed `fatal_timeout` and crash the server. (We simplified the code for illustration)

any quantitative characteristic results. Finally, we also warn the readers to take our findings together with above methodology and selected applications.

3. Bug pattern study

Different bug patterns usually demand different detection and diagnosis approaches. In Table 4, we classify the patterns of the examined non-deadlock concurrency bugs into three categories: Atomicity, Order, and Other, which are described in Table 2. Note that the categories are distinguished from each other by the root cause of a bug, regardless of the possible bug fix strategies.

Application	Total	Atomicity	Order	Other
MySQL	14	12	1	1
Apache	13	7	6	0
Mozilla	41	29	15	0
OpenOffice	6	3	2	1
Overall	74	51	24	2

Table 4. Patterns of non-deadlock concurrency bugs. (There are 3 examined bugs, whose patterns can be considered as either atomicity or order violation. Therefore, they are considered in both categories.)

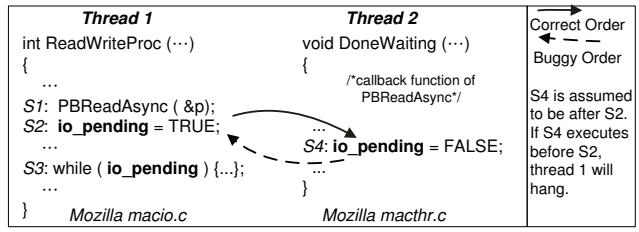

Figure 4. A write-write order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 is expected to write `io_pending` to be FALSE some time after thread 1 initializes it to be TRUE. Note that, this bug could be fixed by making S1 and S2 atomic. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

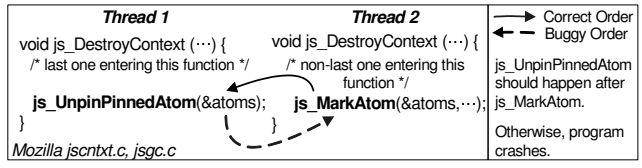

Figure 5. A Mozilla bug that violates the intended order between two groups of operations.

Finding (1): Most (72 out of 74) of the examined non-deadlock concurrency bugs are covered by two simple patterns: *atomicity-violation* and *order-violation*.

Implications: Concurrent program bug detection, testing and language design should first focus on these two major bug patterns.

The Finding (1) can be explained by the fact that programmers generally put their intentions on atomic regions and execution orders, but it is not easy to enforce all these intentions correctly and completely in implementation.

Since programmers think sequentially, they tend to assume that small code regions will be executed atomically. For example, in Figure 1, programmers assume that if S1 reads a non-NULL value from `thd->proc_info`, S2 will also read the same value. However, such an *atomicity assumption* can be violated by S3 during concurrent execution, and it leads to a program crash.

It is also common for programmers to assume an order between two operations from different threads, but programmers may forget to enforce such an order. As a result, one of the two operations may be executed faster (or slower) than the programmers' assumption, and it makes the order bug manifest. In the Mozilla bug shown in Figure 2, it is easy for programmers to assume wrongly that thread 2 would dereference `mThread` *after* thread 1 initializes it, because thread 2 is created by thread 1. However, in real execution, thread 2 may be very quick and dereference `mThread` *before* `mThread` is initialized. This unexpected order leads to program crash. Note that even though the bug can be fixed with locks, the root cause of the bug is not an atomicity violation, but an order violation.

Concurrency bugs violating other types of programmers' intentions also exist, but are much rarer as shown in Table 4. Figure 3 shows an example. In one version of MySQL, programmers use a timeout threshold `fatal_timeout` to detect deadlock. The server will crash, if any thread waits for a lock for more than `fatal_timeout` amount of time. However, when programmers set the threshold, they under-estimate the workload. As a result, users found that the MySQL server keeps crashing under heavy workload (with 2048 worker-thread setting). Such a performance-related as-

sumption is neither atomicity intention nor order intention. This bug is fixed by limiting the number of worker-threads.

Finding (2): A significant number (24 out of 74) of the examined non-deadlock concurrency bugs are order bugs, which are *not* addressed by previous bug detection work.

Implications: New bug detection techniques are desired to address order bugs.

As we discussed above, it is common for programmers to assume a certain order between two operations from two threads. Specifically, programmers can have an order intention i) between a write and a read (Figure 2) to one variable; ii) between two writes (Figure 4) to one variable; or iii) between two groups of accesses to a group of variables (Figure 5). In Figure 4, programmers expect S2 to initialize `io_pending` before S4 assigns a new value, `FALSE`, to it. However, the execution of the asynchronous read can be very quick and S4 may be executed before S2, contrary to the expectation of programmers. This makes thread 1 to hang. In another example shown in Figure 5, `js_UnpinPinnedAtom` frees all elements in the `atoms` array. This set of memory accesses to the whole array is expected to happen after `js_MarkAtom`, which may access some elements in `atoms`.

Note that the above order bugs are *different* from data race bugs and atomicity violation bugs. Even if two memory accesses to the same variable are protected by the same lock or two conflicting code regions are atomic to each other, the execution order between them still may not be guaranteed. We should also note that some order-violation bugs could be *fixed* using coarser-grained locking, as in example Figure 2 and Figure 4; some others cannot be fixed by locks, as in example Figure 5 and Figure 7 (will be discussed later). This is not related to the bug root cause, and does not affect our bug pattern classification.

Although important and common, order-violation bugs have not been well studied by previous research. Many order bugs will be missed by existing concurrency bug detectors, which mainly focus on race bugs or atomicity bugs. New techniques are desired for solving the order problems.

4. Bug manifestation study

Manifestation condition of a concurrency bug is usually a specific order among a set of memory accesses or system events. In this section, we study the characteristics of real world concurrency bug manifestation, following the methodology defined in Table 2. We will discuss guidance for concurrent program testing and concurrency bug detection based on our observations.

4.1 How many threads are involved?

Finding (3): The manifestation of most (101 out of 105) examined concurrency bugs involves no more than two threads.

Implications: Concurrent program testing can pairwise test program threads, which reduces testing complexity without losing bug exposing capability much.

Finding (3) tells us that even though the examined server programs use hundreds of threads, in most cases, only a small number (mostly just *two*) of threads are involved in the manifestation of a concurrency bug.

The underlying reason for this is that most threads do *not* closely interact with many others, and most communication and collaboration is conducted between two or a small group of threads. As a result, manifestation conditions of most concurrency bugs do not involve many threads. For examples, all of the bugs presented in Section 3, except the one shown in Figure 3, are guaranteed to manifest if their execution follow certain partial orders (marked by dotted lines in the figures) between two threads.

Non-deadlock concurrency bugs					
Application	Total	Env.	>2 threads	2 threads	1 thread
MySQL	14	1	1	12	0
Apache	13	0	0	13	0
Mozilla	41	1	0	40	0
OpenOffice	6	0	0	6	0
Overall	74	2	1	71	0
Deadlock concurrency bugs					
Application	Total	Env.	>2 threads	2 threads	1 thread
MySQL	9	0	0	5	4
Apache	4	0	0	4	0
Mozilla	16	0	1	14	1
OpenOffice	2	0	0	0	2
Overall	31	0	1	23	7

Table 5. The number of threads (or environments) involved in concurrency bugs.

We should note that this finding is *not* opposite to the common observation that concurrency bugs are sometimes easier to manifest at a heavy-workload (concurrent execution of many threads). In many cases, the manifestation condition involves only two threads. Heavy-workload increases the resource competition and context switch intensity. It therefore increases the possibility of hitting certain orders among the *two threads* that can trigger the bug. The manifestation condition still involves just two threads.

Our finding implies that testing can focus on execution orders among accesses from every pair of threads. Such pairwise testing technique can prevent the testing complexity from increasing exponentially with the number of threads. At the meantime, few concurrency bugs would be missed.

There are also cases where the bug manifestation relies on not only memory accesses within the program, but also environmental events (as shown in column 'Env' in Table 5). For example, one Mozilla bug cannot be triggered unless another program modifies the same file concurrently with Mozilla. Exposing such bugs needs special system support.

Finding (4): The manifestation of some (7 out of 31) deadlock concurrency bugs involves only one thread.

Implications: This type of bug is relatively easy to detect and avoid. Bug detection and programming language techniques can try to eliminate these simple bugs first.

It usually happens when one thread tries to acquire a resource held by itself. Detecting and analyzing this type of bugs are relatively easy, because we do not need to consider the contention from other concurrent execution components.

4.2 How many variables are involved?

Are concurrency bugs synchronization problems among accesses to one variable or multiple variables? To answer this question, we examine the number of variables (or resources) involved in the manifestation of each concurrency bug. The examination result is shown in Table 6.

Finding (5): 66% (49 out of 74) of the examined non-deadlock concurrency bugs involve only one variable.

Implications: Focusing on concurrent accesses to one variable is a good simplification for concurrency bug detection.

Non-deadlock concurrency bugs			
Application	Total	>1 variables	1 variable
MySQL	14	6	8
Apache	13	4	9
Mozilla	41	15	26
OpenOffice	6	0	6
Overall	74	25	49

Deadlock concurrency bugs				
Application	Total	>2 resources	2 resources	1 resource
MySQL	9	0	5	4
Apache	4	0	4	0
Mozilla	16	1	14	1
OpenOffice	2	0	0	2
Overall	31	1	23	7

Table 6. The number of variables (resources) involved in concurrency bugs.

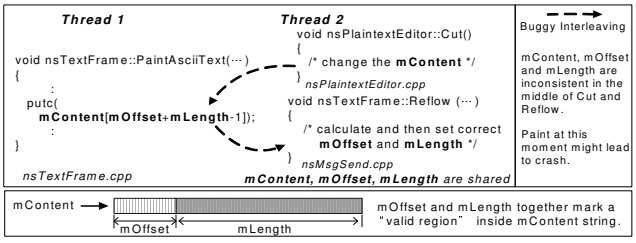

Figure 6. A multi-variable concurrency bug from Mozilla. Accesses to three correlated variables, `mContent`, `mOffset` and `mLength`, should be synchronized.

Finding (5) confirms our intuition. Flipping the order of two accesses to different memory locations does not *directly* change the program state, and therefore is less likely to cause problems. Figure 1, 2, and 4 are all examples of single variable concurrency bugs: their manifestation can be guaranteed by certain order among accesses to one variable.

This finding supports the single-variable assumption taken by many existing bug detectors. For example, data race bug detection [37, 42] checks the synchronization among accesses to one variable; some atomicity violation bug detection tools also focus on atomic regions related to one variable [23, 41].

Finding (6): A non-negligible number (34%) of non-deadlock concurrency bugs involve more than one variable.

Implications: We need *new* concurrency bug detection tools to address multiple variable concurrency bugs.

Multiple variable concurrency bugs usually occur when unsynchronized accesses to correlated variables cause inconsistent program state. Semantic connections among variables are common, and therefore, multiple variable concurrency bugs are common too.

Figure 6 shows an example of multiple variable concurrency bug from Mozilla. In this example, `mOffset` and `mLength` together mark the region of useful characters stored in dynamic string `mContent`. Thread 1 and 2's concurrent accesses to these *three* variables should be synchronized, otherwise thread 1 might read inconsistent values and access invalid memory address. Here, controlling the order of memory accesses to any single variable, *cannot* guarantee the bug to manifest. For example, it is *not* wrong for thread 1 to read `mContent` either before or after thread 2's modification to all of *three* variables. The required condition for the bug manifestation is that thread 1 uses the *three* correlated variables in the middle of thread 2's modification to these *three* variables.

As discussed above, most existing bug detection tools only focus on single-variable concurrency bugs. Although this simplification provides a good starting point for concurrency bug detection, future research should not ignore the problem of multi-variable concurrency bugs.

The difficulty of detecting multiple variable concurrency bugs is that it is hard to infer which accesses, to different variables, should be well synchronized. Solving this problem will not only benefit automatic concurrency bug detection, but also provide useful hints for programmers to specify correct transactions or atomic regions for transactional memory or atomicity bug detection tools [12].

Finding (7): 97% (30 out of 31) of the examined deadlock concurrency bugs involve at most two resources.

Implications: Deadlock-oriented concurrent program testing can pairwise test the order among acquisition and release of two resources.

Among the examined deadlock bugs, only one bug is triggered by three threads circularly waiting for three resources. Leveraging this finding, pairwise testing on resources can prevent the testing complexity from increasing exponentially with the total number of resources.

4.3 How many accesses are involved?

We find that the manifestation of most concurrency bugs involves only two threads and a small number of variables. However, the number of accesses from one thread to each variable can still be huge. Therefore, we need to study how many accesses are involved in the bug manifestation.

Non-deadlock concurrency bugs						
Application	Total	1 acc.*	2 acc.	3 acc.	4 acc.	>4 acc.
MySQL	14	0	2	7	4	1
Apache	13	0	6	5	2	0
Mozilla	41	0	12	18	5	6
OpenOffice	6	0	2	3	1	0
Overall	74	0	22	33	12	7
Deadlock concurrency bugs						
Application	Total	1 acc.*	2 acc.	3 acc.	4 acc.	>4 acc.
MySQL	9	4	1	4	0	0
Apache	4	0	0	4	0	0
Mozilla	16	1	2	12	0	1
OpenOffice	2	2	0	0	0	0
Overall	31	7	3	20	0	1

Table 7. The number of accesses (or resource acquisition/release) involved in concurrency bugs. (*: "1 acc." case happens only in deadlock bugs, when one thread waits for itself. The bug triggering therefore does not depend on any inter-thread order problem.)

Finding (8.1): 90% (67 out of 74) of the examined non-deadlock bugs can deterministically manifest, if certain orders among at most four memory accesses are enforced.

Finding (8.2): 97% (30 out of 31) of the examined deadlock bugs can deterministically manifest, if certain orders among at most four resource acquisition/release operations are enforced.

Implications: Concurrent program testing can focus on the partial order among every small groups of accesses. This simplifies the interleaving testing space from exponential to polynomial regarding to the total number of accesses, with little loss of bug exposing capability.

The Finding (8.1) can be easily understood, considering that most of the examined concurrency bugs have simple patterns and involve a small number of variables. Most of the exceptions come from those bugs that involve more than two threads and/or more than two variables.

The Finding (8.2) is also natural, considering that most of our examined deadlock bugs involve only two resources.

The above findings have significant implication for concurrent program testing. The challenge in concurrent program testing is that the number of all possible interleavings is exponential to the number of dynamic memory accesses, which is too big to thoroughly explore. Our finding provides support to a more effective design of interleaving testing [21]: exploring all possible orders within every small groups of memory accesses, e.g. groups of 4 memory accesses. The complexity of this design is only polynomial to the number of dynamic memory accesses, which is a huge reduction from the exponential-sized all-interleaving testing scheme. Furthermore, the bug exposing capability of this design is almost as good as exploring all interleavings. It would miss only few bugs in our examination.

A recent model checking work [28] uses the heuristic to start the checking from interleavings with small numbers of context switches. Our study provides support for this heuristic.

Of course, enforcing a specific partial order among a set of accesses is not trivial. The program input and many accesses need to be controlled to achieve that. How to leverage our finding to enable practical and powerful concurrent program testing and model checking remains as future work.

5. Bug fix study

5.1 Fix strategies

Before we check how the real world bugs were fixed, our guess was that adding or changing locks should be the most common way to fix concurrency bugs. However, the characteristic result is contrary to our guess, as shown in Table 8.

Application	Total	COND	Switch	Design	Lock	Other
MySQL	14	2	0	5	4	3
Apache	13	4	2	3	4	0
Mozilla	41	13	8	9	9	2
OpenOffice	6	0	0	2	3	1
Overall	74	19	10	19	20	6

Table 8. Fix strategies for non-deadlock concurrency bugs (all categories are explained in Table 2).

Finding (9): Adding or changing locks is *not* the major fix strategy. It is used for only 20 out of 74 non-deadlock concurrency bugs that we examined.

Implication: There is no silver bullet for fixing concurrency bugs. Just telling programmers that certain conflicting accesses are not protected by the same lock is not enough to fix concurrency bugs.

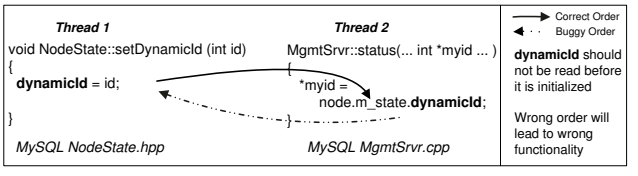

Figure 7. A MySQL bug that cannot be fixed by adding/changing locks.

There are two reasons for this controversy. First of all, locks cannot guarantee to enforce some synchronization intentions, such as A should happen before B. Therefore, adding/changing locks can not fix certain types of bugs. Figure 5 shows such an exmaple. Here we show another simple example in Figure 7. Secondly, even if adding/changing locks can fix a bug, in many cases, it is not the best strategy, because it may hurt the performance or introduce new bugs, such as deadlock bugs.

In the following, we describe the different strategies, other than adding/changing locks, used by programmers. We will see that these strategies usually require deep understanding of program semantics. At the mean time, they usually have better performance than corresponding lock-based fixes, if existing.

(1) Condition check (denoted as COND). Condition check can be used in different ways to help fix concurrency bugs. One way is to use while-flag to fix order-related bugs, such as the bug shown in Figure 5. The other way is to add consistency check to monitor the bug-related program states. This enables the program to detect buggy interleavings and restore program states. For example, to fix the bug shown in Figure 6, the program does consistency check `if(strlen(mContent)>= mOffset+mLength)` before it executes `putc` function. The `putc` will be skipped if the consistency check fails. In another example shown in Figure 8, condition `(n!=block->n)` is checked to see whether the shared variable `block->n` has been overwritten since the last time it was read. If `n` is not consistent with `block->n`, the program rolls back and reads `block->n` again. Note that, above fix strategy does *not* eliminate the buggy interleaving, which is usually the purpose of lock-based fixes. Instead, it focuses on detecting buggy interelavings and makes sure the program states corrupted by the buggy interelavings can be recovered in time. It has better performance than corresponding lock-based fixes.

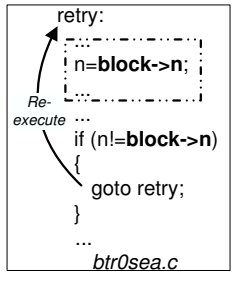

Figure 8. A MySQL bug fix.

(2) Code switch (denoted as Switch). Switching the order of certain code statements can fix some order-related bugs. For example, the order bug shown in Figure 4 is fixed by switching statements S1 and S2, so that S2 is always executed before S4.

(3) Algorithm/Data-structure design change (denoted as Design). This includes different types of algorithm changes and data structure changes that help to achieve correct synchronization. Some design changes are simple, just modifying a few data structures. For example, in the MySQL bug #7209, the bug is caused by unprotected conflicting accesses to a shared variable `HASH::current_record`. Programmers recognize that this variable does not need to be shared. They simply move the field `current_record` out of the class `HASH`, making it a local variable for each thread, and fix the bug. As another example, in Mozilla bug #201134, one thread needs to conduct a series of operations on a shared variable `nsCertType`. In order to enforce the atomicity of that series of operations, programmers simply let program read `nsCertType` into a local variable, conduct operations on the local variable, and store the value back to `nsCertType` at the end. Some design changes are more complicated, involving algorithm re-design. For example, in Mozilla bug #131447, programmers changed a message handling and queueing algorithm to tolerate special timing when a reply message arrives before its corresponding callback function is ready.

As we can see, fixing concurrency bugs is much more complicated than just adding or changing lock operations. Race detection tools can help programmers conduct those lock-related fixes, but this is not enough. It is desired to have more tools to help programmers figure out the bug pattern, the consistency condition associated with each bug, etc. For example, if programmers know that the bug is an order-violation bug and they also know what the consistency condition is, it is easy to come out with a *condition check* fix. This is the challenge for future research on concurrency bug detection and diagnosis.

Application	Total	GiveUp	Split	AcqOrder	Other
MySQL	9	5	0	2	2
Apache	4	2	0	2	0
Mozilla	16	11	1	3	1
OpenOffice	2	1	0	0	1
Overall	31	19	1	7	4

Table 9. Fix strategies for deadlock bugs (all categories are explained in Table 2)

Finding (10): The most common fix strategy (used in 19 out of 31 cases) for the examined deadlock bugs is to let one thread give up acquiring one resource, such as a lock. This strategy is simple, but it may introduce other non-deadlock bugs.

Implication: We need to pay attention to the correctness of some "fixed" deadlock bugs.

In many cases, programmers find it unnecessary or not worthwhile to acquire a lock within certain program context. Therefore, they simply drop the resource acquisition to avoid the deadlock.

However, this strategy could introduce non-deadlock concurrency bugs. In some of our examined bug reports, programmers explicitly say that they know the fix would introduce a new non-deadlock concurrency bug. They still adopt the fix, because they gamble that the probability for the non-deadlock bug to occur is small. In the future, techniques combining optimistic concurrency and rollback-reexecution, such as TM, can help fix some deadlock bugs. Of course, using these techniques should also be careful, because they might introduce live-lock problems.

5.2 Mistakes during bug fixing

Fixing bugs is hard. Some patches released by programmers are still buggy. In order to investigate the nature of buggy patches, we collect all the distinct buggy patches of the 57 Mozilla concurrency bugs [1]. Specifically, we first gather all the intermediate (non-final) patches submitted by Mozilla programmers for these 57 bugs. We then manually check each patch and filter out non-bug-fixing patches, which only change comments or code structures for maintenance purpose.

Our study finds that 17 out of the 57 Mozilla bugs have at least one buggy patches. On average, 0.4 buggy patches were released before every final correct patch. Among all the 23 distinct buggy patches, 6 of them only decrease the occurrence probability of the original concurrency bug, but fail to fix the original bug completely (an example is shown in Figure 9). 5 of them introduce new concurrency bugs. The other 12 introduce new non-concurrency bugs. Programmers need help to improve the quality of their patches.

5.3 Discussion: bug avoidance

Good programming languages should help avoid some bugs during implementation. Transactional memory (TM) is a popular trend of

[1] We focus on Mozilla, because it has the best maintenance of patch update information.

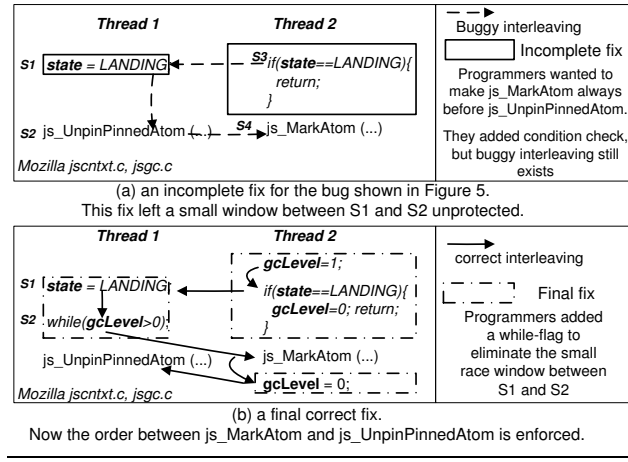

Figure 9. The process of fixing the bug shown in Figure 5. Programmers want to make sure `js_MarkAtom` will not be called after `js_UnpinPinnedAtom`. They first added a flag variable `state` to fix the bug. However, that left a small window between S1 and S2 unprotected. They finally added a second flag variable `gcLevel` to completely fix the bug.

programming language feature for easing concurrent programming. To estimate its benefit and what more are needed along this direction, we study the 105 concurrency bugs to see how many of them can potentially be avoided with TM support. Furthermore, we study what are the issues that future concurrent programming language design needs to address.

Again, our analysis should be interpreted with our examined applications and evaluation methodology in mind, as discussed in Section 2.3. In addition, since different TM designs may have different features, in our discussion, we focus on the basic atomicity and isolation properties of TM. We discuss the benefits and concerns in general, based on such basic TM designs [2, 16, 25, 26]. It is definitely possible for advanced TM designs to address some of the concerns we will discuss, which is exactly the purpose of our discussion: provide more real-world information and help improve the design of TM.

Application	Total	Can Help	TM might help(concerns:)			Little Help
			Long	Rollback	Nature	
MySQL	23	7	0	14	0	2
Apache	17	7	0	3	1	6
Mozilla	57	25	8	9	5	10
OpenOffice	8	2	0	4	0	2
Overall	105	41	8	30	6	20

Table 10. Can TM help avoid concurrency bugs?

Finding (11): TM can help avoid many concurrency bugs (41 out of the 105 concurrency bugs we examined).

Implication: Although TM is not a panacea, it can ease programmers correctly expressing their synchronization intentions in many cases, and help avoid a big portion of concurrency bugs.

Atomicity violation bugs and deadlock bugs with relatively small and simple critical code regions can benefit the most from TM, which can help programmers clearly specify this type of atomicity intention. Figure 8 shows an example, where programmers use a consistency check with re-execution to fix the bug. Here, a *transaction* (with abort, rollback and replay) is exactly what programmers want.

Finding (12): TM can potentially help avoid many concurrency bugs (44 out of the 105 concurrency bugs we examined), if some concerns can be addressed, as shown in Table 10.

Implication: TM design can combine system supports and other techniques to solve some of these concerns, and further ease the concurrent programming.

One concern, not a surprise, is I/O operations. As operations like I/O are hard to roll back, it is hard to use TM to protect the atomicity of code regions which include such operations. Take the concurrency bug in Figure 1 as an example. Since S2 involves a file operation, TM might need non-trivial undo techniques to protect the S1–S2 atomic region.

Other concerns, such as atomic region size and special code nature, also exist. For example, the atomic code regions of several Mozilla bugs include the whole garbage collection process. These regions could have too large memory footprint to be effectively handled by hardware-TM.

Many of the above concerns are addressable in TM, but with higher overhead and complexity. For example, some of the rollback concerns can be addressed using system supports. Very long transactions can be addressed by combining software and hardware TMs.

Finding (13): 20 out of the 105 concurrency bugs that we examined cannot benefit from the basic TM designs, because the violated programmer intentions, such as order intentions, cannot be guaranteed by the basic TM.

Implications: Apart from atomicity intentions, there is also a significant need for concurrent programming language features to help programmers express order intentions easily.

Programmers' order intention is the major type of intention that cannot be easily enforced by the basic TM design or locks. In general, the basic TM designs cannot help enforce the intention that A has to be executed before B. Therefore, they cannot help avoid many related order-violation bugs [2]. Among all order-violation bugs, we find a sub-type of order intentions that are extremely hard to be enforced by basic TM designs: A must be either executed before B or not executed at all. In other words, programmers do not want B to wait for A. They simply skip A if B is already executed. For example, in one Mozilla bug, thread 1 keeps inserting entities to a cache and thread 2 would destroy the cache at some moment. Based on the description in the bug report, programmers do not want thread 2 to wait for thread 1 to finish all insertions. The program simply skips any insertion attempt after the cache is destroyed. This happens for 7 bugs.

In order to help avoid above 20 bugs, the semantic design, instead of implementation schemes, of the basic TM needs to be enhanced. Recently, some TM designs [5, 17] are equipped with rich semantics (such as watch/retry, retry/orElse) and can help enforce some of the above synchronization intentions. We hope our bug characteristic study can help future research to decide the best TM design.

6. Other characteristics

Bug impacts: Among our examined concurrency bugs, 34 of them can cause program crashes and 37 of them cause program hangs. This validates that concurrency bug is a severe reliability problem.

[2] Some order-violation bugs can be avoided by TM. In those cases, order intentions can be enforced as side effects while TM enforces the atomicity of related code regions (an example is shown in Figure 2).

Some concurrency bugs are very difficult to repeat. In one bug report (Mozilla#52111), the reporter complained that "I develop Mozilla full time all day, and I get this bug only once a day". In another bug report (Mozilla#72599), the reporter said that "I saw it only once ever on *g* (never on other machines). Perhaps the dual processor of *g* makes it occur."

Test cases are critical to bug diagnosis. Programmers' discussions show that a good test case to repeat a concurrency bug is very important for diagnosis. In Mozilla bug report #73291, the programmers once gave up on this bug and closed the bug report, because they could not repeat the bug. Fortunately, somebody worked out a way to reliably repeat the bug, and the bug was fixed subsequently. In another Mozilla bug report (Mozilla#72599), the programmers finally gave up repeating the bug and simply submit a patch based on their "guessing", and this led to a wrong fix.

Programmers lack diagnosis tools. From the bug reports, we notice that many concurrency bugs are diagnosed simply by programmers reading the source code. For example, for 29 out of the 57 Mozilla bugs, the bug reports did not mention that the programmers ever leveraged any information from any tools, core dumps, or stack traces, etc. Sometimes programmers tried `gdb`, but could not get useful information. We have never seen programmers mentioned that they used any automatic diagnosis tools. In contrast, in many bug reports about memory bugs, programmers mentioned that they got help from Valgrind, Purify, etc [20].

7. Related work

Bug characteristic studies A lot of work has been done to study the bug characteristics in large software systems. Many of them provide precious information to help improve software reliability from different aspects, such as bug detection [8, 38], fault tolerance [14], failure recovery [6], fault prediction and testing [32], etc. In a recent work [43], people also studied how the recent trends (availability of commercial tools, open-source, etc) affect the general bug characteristics (bug distribution, fixing time) for all bugs.

Unfortunately, few previous work have studied concurrency bugs, probably because real world concurrency bugs are hard to collect and analyze. For example, in a previous study [6], only 12 concurrency bugs were collected from three applications: MySQL, GNOME and Apache. Under this situation, a previous concurrency bug pattern study [11] had to ask students to purposely write concurrent programs containing bugs, which cannot well represent the real world bug characteristics. Unlike previous work, we study the bug pattern, manifestation, and fix of 105 real world concurrency bugs from 4 large open source applications. Our study provides many findings and implications for addressing the correctness problems in concurrent programming.

Improving concurrent program reliability Techniques to improve concurrent program quality is related to our work. Due to space limit, here we briefly discuss the work that have not been discussed in previous sections.

In software testing, people proposed different coverage criteria in order to selectively test concurrent program interleavings. Unfortunately, these proposals are either too complicated [39] or based on heuristics [4, 9]. Our study of concurrency bug manifestation can help understand the trade-off between testing complexity and bug exposing capability and help design better coverage criteria.

In programming language area, designs other than transactional memory are also studied. AtomicSet [40] associates synchronization constraints with data instead of code region. This design can help avoid some multiple variable related concurrency bugs. Autolocker [24] eases programmers specifying atomic regions by automatically assigning locks. Our characteristics study provides more motivation for these new language features.

8. Conclusions and future work

This paper provides a comprehensive study of the real world concurrency bugs, examining their pattern, manifestation, fix strategy and other characteristics. Our study is based on 105 real world concurrency bugs, randomly collected from 4 representative open-source programs: MySQL, Apache, Mozilla, and OpenOffice. The result of our study includes many interesting findings and implications for concurrency bug detection, testing and concurrent programming language design. Future research can benefit from our study in various aspects. For example, future work can design new bug detection tools to address multiple-variable bugs and order-violation bugs; can pairwisely test concurrent program threads and focus on partial orders of small groups of memory accesses to make the best use of testing effort; can have better language features to support "order" semantics to further ease concurrent programming. In the future, we will extend our study with other types of real world applications.

9. Acknowledgments

We thank the anonymous reviewers for useful feedback, the Opera groups and Raluca A. Popa for useful discussions and paper proofreading. This research is supported by NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), DOE Early Career Award DE-FG02-05ER25688, Intel gift grants, Korea Foundation for Advanced Studies (KFAS) doctorial scholarship, and the Information and Telecommunication National Scholarship program of the Ministry of Information and Communication of Korea.

References

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Transactional programming in a multi-core environment. In *PPOPP*, 2007.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[4] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPoPP*, 2005.

[5] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *PLDI '06*, 2006.

[6] S. Chandra and P. M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN*, 2000.

[7] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[9] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multi-threaded java program test generation. *IBM Systems Journal*, 2002.

[10] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[11] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, 2003.

[12] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[13] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, 1997.

[14] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *DSN*, 2003.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[16] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[17] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05*, 2005.

[18] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix*, 1992.

[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in o perating system code. In *OSDI*, 2004.

[20] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability (ASID'06)*, 2006.

[21] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *FSE*, 2007.

[22] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP07*, 2007.

[23] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[24] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, 2006.

[25] M. Moir. Transparent support for wait-free transactions. In *11th International Workshop on Distributed Algorithms*, 1997.

[26] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[27] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 2006.

[28] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[29] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, 2002.

[30] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *ENTCS*, 2003.

[31] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP*, 1991.

[32] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *TSE*, 2005.

[33] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, 2003.

[34] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.

[35] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies c a safe method to survive software failures. In *SOSP*, 2005.

[36] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: transactional memory for an operating system. In *ISCA*, 2007.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[38] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *FTCS*, 1992.

[39] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 1992.

[40] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[41] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[42] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[43] Z. Li et. al. Have things changed now? – an empirical study of bug characteristics in modern open source software. In *ASID workshop in ASPLOS*, 2006.