A Study of the Internal and External Effects of Concurrency Bugs

Pedro Fonseca, Cheng Li, Vishal Singhal*, and Rodrigo Rodrigues
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Concurrent programming is increasingly important for achieving performance gains in the multi-core era, but it is also a difficult and error-prone task. Concurrency bugs are particularly difficult to avoid and diagnose, and therefore in order to improve methods for handling such bugs, we need a better understanding of their characteristics. In this paper we present a study of concurrency bugs in MySQL, a widely used database server. While previous studies of real-world concurrency bugs exist, they have centered their attention on the causes of these bugs. In this paper we provide a complementary focus on their effects, which is important for understanding how to detect or tolerate such bugs at runtime. Our study uncovered several interesting facts, such as the existence of a significant number of latent concurrency bugs, which silently corrupt data structures and are exposed to the user potentially much later. We also highlight several implications of our findings for the design of reliable concurrent systems.

1 Introduction

We are witnessing an unprecedented rise in the parallelism of computer systems. The number of cores in commodity processors has been steadily increasing. Today, dual-core and quad-core processors are commonplace; Intel has recently announced an 8-core processor [3], and specialized CPUs with even more cores are currently being manufactured [1, 2]. However, increasing the number of processors is not the way by which CPU manufacturers have traditionally increased the performance of their hardware. Clock speeds no longer increase at a significant rate; as a result, software no longer automatically runs significantly faster as new chips are deployed. Consequently, for software to extract performance gains out of the extra processing capacity, programmers will have to design their software in a more parallel way.

However, parallel programming is challenging and often error prone. It is difficult enough for programmers to reason about all the possible inputs and the flow of execution in single-threaded applications; reasoning about all the different thread interleavings that can occur in concurrent programs, combined with all possible inputs, is even more difficult. Additionally, the non-determinism that is inherent to concurrency bugs, which are only triggered under certain thread interleavings, makes it difficult to reproduce, identify, analyze, or correct such programming mistakes. But at the same time this non-determinism can be essential in handling them (e.g., using fault detection, fault tolerance, or fault recovery) since it enables exploring redundant, diverse executions using different thread interleavings.

To improve 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 [11,14,22], 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 [27].

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 (i.e., a Byzantine failure), in order to gauge the effectiveness of using multi-threaded replicas to ensure fault diversity in a Byzantine-fault-tolerant replication scheme [10].

In this paper we provide the complementary angle of studying the *effects* of concurrency bugs that affect parallel applications. In particular, we exhaustively study real

*Currently a student at BITS Pilani, India. Work done during his internship at MPI-SWS.

978-1-4244-7499-8 /10/$26.00 ©2010 IEEE

concurrency bugs that were found in MySQL [5], 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 [13], 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. We characterize Byzantine failures that are caused by 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 bugs that cause Byzantine failures 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 1.

The remainder of the paper is organized as follows. In Section 2 we describe our methodology. We then present an overview of the MySQL application in Section 3. The results of our study are presented in Section 4 and in Section 5 we discuss their implications. We survey related work in Section 6 and we conclude in Section 7.

2 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 [22].

2.1 Choice of concurrent application

We selected MySQL as the target of our study for three main reasons. First, it is a widely deployed database. Databases are a critical component of the IT infrastructure of many corporations, and MySQL represents a substantial share of that market (about $1/3$ of deployed database systems [4]). This implies that there is market pressure for a quality development and maintenance process, so this is an instance of well-maintained software where finding and eliminating bugs matters. Second, it is an open source application with a well-maintained bug report database. Having access to the source code and the bug logs is necessary for an in-depth analysis. Finally, it is a highly concurrent application with rich semantics, and it has a large code base. These characteristics make MySQL representative of some of the biggest challenges that we will be facing as complex applications become more and more concurrent.

In Section 3 we provide some brief background on MySQL, which will help in better understanding our results.

2.2 Concurrency bug selection

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.

We searched the MySQL bug report database for bugs that contained keywords commonly associated with concurrency bugs. Such keywords included the following terms: *lock*, *acquire*, *compete*, *atomic*, *concurrency*, *synchronization*, etc. In addition to this 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 bug reports are not likely to have detailed discussions and more importantly, in general, they won't 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 [26].

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 (defined in Section 3) and so we also excluded them. In addition, we excluded bugs for which the bug log 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 studies [11,22].

Table 2 shows the bug count across the different stages of the bug selection process.

Note that this selection process 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 dif-

[1] The term *latent bug* is used in other papers [8,18,20] 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 shows the 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 – we also need to handle non-deadlock bugs.
We found a significant fraction of semantic/Byzantine bugs (15%).	Techniques for Byzantine fault tolerance 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 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 1. Main findings of this study and their implications. The methodology for collecting the data presented here is described in Section 2 and the results are explained in detail in Section 4.

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

Table 2. Bug counts for different stages of the analysis.

ficult 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.

2.3 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 debuggers 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 when they are what are their effects [2]. In addition, some of this information was also used to estimate the complexity of fixing concurrency bugs and their severity.

3 MySQL

In this section we provide a brief overview of the characteristics of MySQL that are relevant for this study.

3.1 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. Here we describe some of the main data structures that will be referred to in later sections.

An important class of stored structures are data files that contain the contents of different tables stored in the database. In addition, a series of files containing the indexes of the tables are also maintained, which allow for fast lookups by the contents of certain columns of the tables.

Another important persistent structure is the binary log (referred to as the binlog structure), which is used for two different purposes. The log is mainly useful when primary-backup replication of the database is used, in which the primary replica writes to the binlog the statements corresponding to all client requests that modify the database state. The backup replicas then sequentially re-execute the statements contained in the binlog. The other use of the binlog is related to other recovery operations such as restoring database state from a backup file, in which case some events that were logged after the backup operation must be re-executed.

Finally, MySQL contains a series of caches that speed up access to persistent structures or processing of requests. For instance, a table cache holds the descriptors of recently accessed tables, while a query cache holds the results of recently executed queries.

3.2 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.

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

Despite the existence of recent proposals for other types of synchronization primitives such as transactional memory [17], there is value in studying and improving the methods that address the problems with lock-based synchronization. This is not only because we still run many applications that use locks, which will benefit from being made more robust for years to come, but also because the vision behind such proposals is not to entirely replace locks, but instead to use these new primitives in smaller sections of the code where the possible performance impact would be lower.

3.3 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 kinds of concurrency bugs are we interested in.

We will only analyze 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 in this study.

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).

4 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 1.

4.1 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

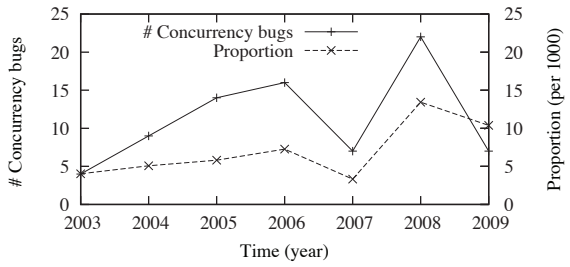

Figure 1. Evolution of bugs (by open date).

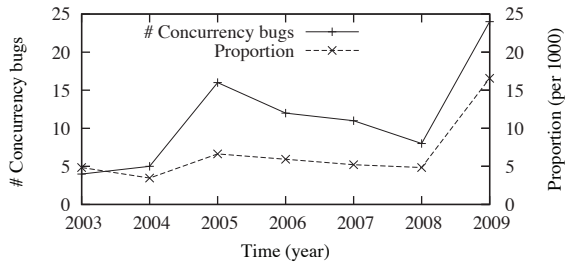

Figure 2. Evolution of bugs (by close date).

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 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 typically require a significant amount of time (i.e., many months) to solve the bugs under analysis. The results are presented in Figures 1 and 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 collect 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 youngest 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 4.7, 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 have not yet been fixed, which means they are not yet closed and were therefore not accounted for in this study.

4.2 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 3. Note that the sum of all occurrences is larger than the total number of bugs because some bugs fit into more than one category.

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 are Byzantine failures, where the application provides the user with a result that violates 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 [10] or run-time verification of the behavior of the application against a specification of the system [30]. We discuss these bugs in more detail in Section 4.4.

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 [22].

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 these are caused by a thread that fails to release a certain lock, causing another thread that tries to

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

Table 3. External effects of concurrency bugs.

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

Table 4. Effects of latent concurrency bugs.

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 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 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 [28]) and deadlocks (e.g., Dimmunix [19]), 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.

4.4 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 [30], which would avoid the use of more expensive techniques such as BFT replication [10].

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 particular 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.

4.5 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.

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.

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 5. Most frequent data structures involved in latent bugs. * The contents of the cache can also be written back to disk.

Next we analyzed whether the data structures involved are persistent structures stored on disk or volatile structures kept in memory. Table 5 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 5 do not add up to the total number of latent bugs because certain bugs affected more than one data structure, as explained before.

4.6 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 6. 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.

4.7 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

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

Table 6. Recovery mechanisms for latent concurrency bugs.

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 7. 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.

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 7. 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 8. 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 do not have a clear explanation, so we defer study of the reasons for this to future work.

Bug immediacy	Time	Patches	Files	Disc.
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	Disc.
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 8. 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.

5 Discussion and limitations

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 [15]) 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 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 work we focused our attention on concurrency bugs found in the MySQL application. A previous paper compared concurrency and non-concurrency bugs of three different database systems including MySQL [32]. 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 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 [22] that compared different applications, about half of the deadlocks found in MySQL involved only 1 resource while almost all of the deadlocks found in Mozilla involved 2 or more resources. 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 [11,22,32]. 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.

6 Related Work

Given the importance of software reliability and the prevalence of bugs in software in general, many studies about bugs have previously been undertaken.

There is a large body of literature about the propagation [33] and even prediction [24] 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 work presented in this paper, which is focused on a specific class of bugs (i.e., concurrency bugs) and on understanding their consequences.

In a previous paper, researchers analyzed the consequences of bugs for three different database systems [32]. However the authors did not distinguish between concurrency and non-concurrency bugs, and only evaluated whether they caused crash or Byzantine faults (since that paper was focused on presenting a replication architecture, instead of being focused on studying bugs). In contrast, we provide a detailed analysis of the effects of the bugs and we focus on concurrency bugs.

Chandra et al. [11] looked at bug databases of three open-source applications (including MySQL) but the focus of their work was quite different from ours. They analyzed all bugs (among which only 12 were concurrent) 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.

Farchi et al. analyzed concurrency bugs, but by artificially creating them [14]. 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.

Recently Lu et al. [22] 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).

There also exist various studies of bug characteristics in software systems focusing on several aspects of generic bugs [12, 16, 21, 25, 31]. In contrast, our study focuses specifically on concurrency bugs, which are more challenging to analyze.

Recently Sahoo et al. have been trying to understand the reproducibility of bugs [29]. 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.

Finally, there exist many proposals for handling concurrency bugs. These represent not only different techniques, but also very different approaches to improving software reliability. They include approaches to avoid bugs [17], to find bugs [13], to mask bugs [32] and even to recover from bugs [9]. Because concurrency bugs, in addition to being dependent on the input, are also dependent on the interleaving chosen by the operating system, there are approaches that specifically handle concurrency bugs by artificially disturbing [6], controlling [23] or limiting [7] thread interleavings. Our work is complementary in that it has the potential to guide and motivate the development of these kinds of techniques and approaches.

7 Conclusion

Concurrency bugs pose a challenge in the development of reliable applications. Concurrency bugs are a type of bug that is likely to become more and more prevalent in the development life cycle as applications become more concurrent to take advantage of parallelism in the hardware.

To gain a better understanding of this problem, we presented a study of concurrency bugs in MySQL. In contrast to previous studies, we focused on the effects of concurrency bugs rather than on their causes.

Studying how bugs manifest enabled us to produce some interesting findings, such as a high prevalence of latent bugs that silently corrupt data structures but may take longer to become externally visible, and a strong correlation between latent bugs and bugs that cause Byzantine failures.

We hope that our study can open interesting avenues for future research. In particular, we intend to develop tools that address the issue of latent bugs from two different angles. First, we need to develop better ways to find these bugs during the course of testing. We intend to develop better tools for catching the subtle corruption of internal state caused by the kinds of bugs we analyzed. Second, latent bugs provide an interesting opportunity to develop techniques that detect them and heal the service state before the buggy output is seen by clients.

Acknowledgments

We are grateful for the feedback provided by the anonymous reviewers. Pedro Fonseca was supported by a grant provided by FCT.

References

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

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

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

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

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

[6] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing scheduling that causes concurrent programs to fail. In *Proc. of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2006.

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

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

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *Proc. of Operating System Design and Implementation (OSDI)*, 2004.

[10] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. of Operating System Design and Implementation (OSDI)*, 1999.

[11] S. Chandra and P. 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*, 2000.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2001.

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

[14] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[15] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of Reliability in Distributed Software and Database Systems*, 1986.

[16] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *Proc. of International Conference on Dependable Systems and Networks*, 2003.

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

[18] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[19] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proc. of Operating System Design and Implementation (OSDI)*, 2008.

[20] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke. Eliminating concurrency bugs with control engineering. *IEEE Computer*, 99(1), 2009.

[21] 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 *Proc. of Architectural and System Support for Improving Software Dependability (ASID)*, 2006.

[22] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGARCH Computer Architecture News*, 36(1):329–339, 2008.

[23] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of Operating System Design and Implementation (OSDI)*, 2008.

[24] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proc. of Conference on Computer and communications security (CCS)*, 2007.

[25] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering (TSE)*, 31(4):340–355, April 2005.

[26] S. Pachev. *Understanding MySQL internals*. O'Reilly Media, Inc., 2007.

[27] S. Park, S. Lu, and Y. 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.

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

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

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

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

[32] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. of Symposium on Operating System Principles (SOSP)*, 2007.

[33] L. Voinea and A. Telea. How do changes in buggy mozilla files propagate? In *Proc. of Symposium on Software Visualization (SoftVis)*, 2006.