What Change History Tells Us about Thread Synchronization*

Rui Gu[3] Guoliang Jin[4] Linhai Song[1] Linjie Zhu[1] Shan Lu[2]

[1]University of Wisconsin-Madison, USA [2] University of Chicago, USA
[3]Columbia University, USA [4] North Carolina State University, USA

ABSTRACT

Multi-threaded programs are pervasive, yet difficult to write. Missing proper synchronization leads to correctness bugs and over synchronization leads to performance problems. To improve the correctness and efficiency of multi-threaded software, we need a better understanding of synchronization challenges faced by real-world developers.

This paper studies the code repositories of open-source multi-threaded software projects to obtain a broad and in-depth view of how developers handle synchronizations.

We first examine how critical sections are changed when software evolves by checking over 250,000 revisions of four representative open-source software projects. The findings help us answer questions like how often synchronization is an afterthought for developers; whether it is difficult for developers to decide critical section boundaries and lock variables; and what are real-world over-synchronization problems.

We then conduct case studies to better understand (1) how critical sections are changed to solve performance problems (i.e. over-synchronization issues) and (2) how software changes lead to synchronization-related correctness problems (i.e. concurrency bugs). This in-depth study shows that tool support is needed to help developers tackle over-synchronization problems; it also shows that concurrency bug avoidance, detection, and testing can be improved through better awareness of code revision history.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability

Keywords

Locks, Empirical Study, Repository Mining, Concurrency Bugs, Performance Bugs, Multi-Threaded Software

*Supported in part by NSF-CCF grants 1439091, 1217582, 1054616; and an Alfred P. Sloan Research Fellowship.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08
http://dx.doi.org/10.1145/2786805.2786815

1. INTRODUCTION

1.1 Motivation

Multi-threaded programs are pervasive, yet difficult to write. In particular, thread synchronization is challenging. Missing proper synchronization causes correctness bugs, such as data races [11, 21, 37, 52, 68] and atomicity violations [12, 13, 30, 63], while over synchronization causes performance problems [7]. Better understanding of real-world synchronization challenges is needed to ease the development process and improve the correctness and efficiency of multi-threaded programs.

To achieve such an understanding, previous studies often resort to open-source bug databases [14, 19, 26, 29]. These databases contain detailed information of reported bugs, including diagnosis discussion, source code, and patches. Such information has enabled previous studies to motivate and guide a wide variety of concurrency-bug research.

Unfortunately, since many real-world software projects are complicated and quickly evolving, a lot of important information, such as the following, is buried in the code revision history and cannot be obtained from bug databases alone:

Information that goes beyond bug reports. Developers conduct synchronization-related code development and maintenance for performance enhancement, functionality changes, readability improvement, and others. These tasks are all important and effort-consuming. They rarely, if ever, get reflected by bug databases. Even for correctness bugs, some of them may be fixed in the code repository but are never reported in bug databases.

Information that is scattered over multiple versions of source code. How is a concurrency bug introduced by code revision? Different patterns could imply different short-cuts for bug avoidance, detection, and testing. How is a critical section formed — is the lock-and-unlock typically introduced together with or after the critical section body? These two different scenarios would demand different tool support for developers. How often are the lock variable and boundaries adjusted for a critical section? Specialized tools may be needed for these adjustments. None of these questions can be answered by checking one version of source code alone or by checking the bug databases alone.

Information that hides within the whole revision history. We have to study a long history to understand trends, like whether the synchronization problem is getting more difficult with software getting more mature and whether a critical section is more likely to change when it ages.

The above information can provide insights for the design of new language features, analysis tools, run-time systems, and code development tools. It can also shed light on new research directions, such as incremental bug detection, synchronization change impact analysis, synchronization change prediction, over-synchronization detection and fixing, and others.

Unfortunately, collecting the above information is challenging. Revision histories of large projects are difficult to study due to their huge volumes. For example, each version of MySQL database contains about four million lines of code. Its repository contains about 7,000 code versions, Facing these many lines of code, a lot of program analysis cannot scale, not to mention manual inspections. Without careful planning and trade-offs, the study will fail.

1.2 Contributions

We study the code repositories of open-source multi-threaded software along several directions to better understand synchronization challenges encountered by developers.

General Study We study how lock-protected critical sections are changed when software evolves. For this study, we design a hierarchical taxonomy for all critical-section changes, based on their structural patterns and purposes.

With this taxonomy in mind, we look at *all* the code changes in the publicly available code repositories of four representative open-source C/C++ software projects. These repositories contain more than 250,000 revisions in total and have 8 – 19 years of code development history. While studying *how many* changes are there under each category, *why* these changes are made, and *when* these changes are made, we have made interesting observations:

- For a notable portion of critical sections (20–25% in our study), the surrounding lock-and-unlock is introduced *after* the enclosed code body. In many cases, the lock synchronization and the code body should have been introduced together, but developers forgot the synchronization. In other cases, software changes bring new and unsafe way of data sharing, and hence demand extra synchronization in old code regions.

- More than three quarters of critical sections in MySQL and Apache code repositories are modified or removed. More than a quarter of critical sections in our study have synchronization adjustments, such as critical section boundary movement and lock variable changes. Developers need tool support to figure out synchronization details, and to keep the synchronization correct and efficient during constant code changes.

- The number of critical section changes remain stable when software ages, but decreases significantly when a critical section ages. Changes that improve performance or fix bugs often happen at a much older age of a critical section than other types of changes.

- Fixing correctness bugs is one of the most common reasons for critical section changes. Enhancing performance is not as common, but still non-negligible, leading to nearly 10% of critical section changes sampled by us.

This study provides motivations, guidelines, and benchmarks for several lines of research, such as synchronization-related change-impact analysis, performance- and correctness-oriented synchronization maintenance (i.e., adjusting existing synchronization locations and variables), over-synchronization detection and fixing, etc. More details are presented in Section 3. All the scripts and results from our study will be released.

In-Depth Case Study Following the above general study, we conduct case studies to better understand *over-synchronization issues* (i.e., unnecessary synchronization degrades execution performance) and *concurrency bug issues* (i.e., lack of or incorrect synchronization hurts execution correctness).

For over-synchronization issues, we manually studied 20 randomly sampled critical-section changes that are used to improve synchronization performance. Our study shows that over-synchronization is a real issue in practice and is cared by real-world developers. When developers change critical sections to alleviate over-synchronization, they struggle with synchronization-correctness reasoning, lock-variable management, and code refactoring. Tool support is needed to help tackle these problems. More details are in Section 4.

For concurrency bug issues, we manually studied the origins of 25 concurrency bugs to see how they are introduced by code revisions. Since this study often requires much deeper understanding of bugs than what change logs provide, some of these 25 bugs are sampled from the change history study mentioned above, and some are from real-world bugs widely used by previous concurrency-bug research [2, 15, 20, 21, 29, 30, 32, 42, 45, 53, 60, 61, 63, 66, 70, 71].

This study reveals interesting findings that can help improve the scalability and accuracy of concurrency-bug detection, including the following, with more details in Section 5.

- Over half of these bugs are introduced under old synchronization contexts (i.e., surrounding locks, preceding barriers/waits, etc.) that are completely unchanged by bug-introducing revisions. This indicates that synchronization analysis can be greatly simplified in concurrency-bug detection and testing when exploiting code history information.

- About half of these bugs only involve shared variables and memory-access instructions that are newly introduced by the bug-introducing revision. This indicates that memory-access analysis can be greatly simplified in concurrency-bug detection and testing when exploiting revision information.

2. METHODOLOGY

This section presents the methodology of our general study about critical-section changes.

Software and Bug Sources We study four representative C/C++ open-source software projects, as shown in Table 1[1]. They are all large and mature projects under active development, with millions lines of code in each version and 8 – 19 years of repository history. Our study looks at *all* the publicly available revisions[2] from their version control systems — SVN for Apache and MPlayer, Mercurial for Mozilla [35], and Bazaar for MySQL [10]. Duplicate revisions, mainly caused by revision merging, are pruned out.

[1] We will refer to "critical section" as *CS*.

[2] We will use *revision* and *version* interchangeably.

Table 1: Applications used in the study (all-*CS*: all unique *CS*es in the repository, each of which could live through many revisions; added-*CS*: *CS*es added since the first version; all-*CS* are more than added-*CS*, as some *CS*es exist in the initial code image; Section 2 discusses how we count *CS*es.)

Application	Repository Info.					Latest Version
	Period	#Rev.	Rev. Size (avg. LoC)	#all-*CS*	#added-*CS*	Size (LoC)
Apache HTTPD Web Server	1996 - 2014	25897	47	366	138	258K
Mozilla Browser Suite	2007 - 2014	188000	78	2898	2036	8.17M
MPlayer Media Player	2001 - 2014	37000	56	76	65	448K
MySQL Database Server	2000 - 2013	6800	494	2095	1548	3.91M

Table 2: Taxonomy of changes ("body": the code region enclosed by lock-and-unlock in a *CS*; "synchronization variable": lock variable; "synchronization primitive": different types of lock operations)

Structural Patterns	
Add	Adding *CS*es
Add_{All}	Synchronization and body added together
Add_{Syn}	Synchronization introduced after body
Rem	Removing *CS*es
Rem_{All}	Synchronization and body removed together
Rem_{Syn}	Synchronization removed alone
Mod	Modifying existing *CS*es
Mod_{Body}	Critical section body modified
Mod_{Syn}	Critical section synchronization modified
Mod_{SynV}	Synchronization variable modified
Mod_{SynP}	Synchronization primitive modified
Mod_{SynB}	Critical section boundary moved
Mod_{SynS}	Critical section split
Mod_{SynU}	Adding unlock operations
Purpose Patterns	
Correctness	Fixing functional bugs
Functionality	Adding or changing code functionality
Maintainability	Code refactoring
Performance	Improving performance
Robustness	Adding sanity checks

Taxonomy Our categorization follows two dimensions — structure and purpose, as shown in Table 2.

Study Mechanisms Accurate categorization requires inter-procedural control-flow and pointer-alias analysis, which unfortunately cannot scale to large code repositories. We choose to use regular-expression based Python scripts, as it offers us the best balance between complexity and accuracy. Alternative approaches like AST-based analysis offers little accuracy increase, while more sophisticated analysis would not scale. Given the complexity constraints, we only consider *CS*es that start and end in the same function.

For each version in the code repository, our script conducts three-step analysis for each line i that appears in the "diff" between this version, referred to as *new* below, and the previous version, referred to as *old* below. The "diff" is obtained through commands like `svn diff`.

First, identifying the innermost enclosing *CS* of i, denoted as C. This is achieved by searching backward and forward from i within the function that contains i, examining every lock-acquisition statement, lock-release statement, as well as the lock variables used by these statements. If no *CS* is found to enclose i, the next two steps are skipped.

Second, identifying *CS*es corresponding to C in the other version (i.e., old or new version depending on which version i is from). To achieve this, we identify all unchanged statements in C and find each such statement's innermost enclosing *CS* in the other version. If multiple unique *CS*es are found, we discover a *CS* split (Mod_{SynS}), a *CS* removal (Rem), or an adding-unlock (Mod_{SynU}). If no *CS* is found, we discover a *CS* addition (Add) or removal (Rem), depending on i is from the new or the old version. If exactly one *CS* C' is found, we go to the third step.

Third, checking the lock-and-unlock operations of C and C' to identify body changes, boundary changes, synchronization primitive changes, lock-variable changes, and so on.

Due to space constraints, some analysis details, such as more detailed categorization and comments handling, are skipped. Manually checking 500 randomly sampled categorization results from our script shows that the false positive rate of our script is below 5%. Our script makes the least accurate categorization for Add_{Syn} and Rem_{Syn}, when insignificant statements appeared before the synchronization enclosing them were added or remained after the synchronization enclosing them were removed.

We use the identity of lock-acquisition function to uniquely identify each *CS*. One lock acquisition followed by multiple releases is counted as one *CS*. We also use the "diff" mechanism to connect the same lock-acquisition function in different versions, so that we can track the changes of one *CS* throughout the software change history.

The lock and unlock primitives considered by our script are obtained by key-word search — lock, latch, and mutex — in the four software projects under study.

About half of Mozilla *CS*es are protected by `AutoLock`, a scope-lock that releases at the end of its current scope. We handle it slightly different from basic locks, and hence have no information about *CS* body change (Mod_{Body}), *CS* split (Mod_{SynS}), and adding unlocks (Mod_{SynU}) for `AutoLock`. We will discuss Mozilla results with and without considering `AutoLock` separately in Section 3.

Threats to Validity Due to the huge amount of code under study, we intentionally trade off some analysis accuracy for analysis speed and hence could miss some *CS* changes. First, *CS*es that start in one function and end in another are not identified. Second, if two pointers $p1$ and $p2$ point to the same lock, a *CS* surrounded by `lock(p1)` and `unlock(p2)` is not recognized; heap-based locks may also cause inaccuracy when we count lock-variable changes (Mod_{SynV})[3]. Third, changes inside functions called by a *CS* are not considered for that *CS*. Furthermore, like all empirical studies, our study cannot cover all software projects in the world. It also cannot cover code changes not committed to the code repository.

[3] Global and heap locks are similarly common in our study.

Even with the above caveats, we believe our study will provide valuable observations and guidelines for future synchronization-related research for two reasons. First, our counting is mostly accurate. The first two issues mentioned above are rare for *CS*es in real-world software. Our manual checking of 500 randomly sampled script results shows that our script has lower than 5% false positive rate. Second, the inaccuracy does not affect the main observations and implications of our study. For example, since we focus on synchronization challenges, the inaccuracy in counting *CS* body changes (e.g., not considering callee changes or not considering body changes in `AutoLock` *CS*es) does not affect our main observations. We could miss some Mod_{SynV} cases due to heap locks, but this does not invalidate our observations, such as "synchronization adjustments are common."

Overall, our study represents our best effort of understanding *CS* related changes in widely used C/C++ open-source software projects. Our methodology takes a trade-off that is suitable for our goal. All results from this study are cross checked by multiple people. All findings below should be interpreted with the above methodology in mind. *We will release all our scripts and results together with the paper.*

3. CRITICAL SECTION CHANGES

3.1 Observations

3.1.1 How Many Changes Are There?

How Many Changes for Each Pattern? The total number of changes ranges from 157 in MPlayer to 7260 in MySQL, as shown in Table 3. The three major patterns, Add, Rem, and Mod, are about equally common. All sub-patterns, except for removing-synchronization-alone (Rem_{Syn}) and *CS*-split (Mod_{SynS}), each contributes to at least 1.5% of all changes and affects at least 3.8% of all *CS*es, as shown in Table 3 and 4. We discuss most common patterns below, and some interesting but less common patterns in Section 3.1.2.[4]

Add, **Add_{All}-vs-Add_{Syn}**: Adding *CS*es contributes to 17–40% of *CS* changes in four projects. The ratio between Add_{All} and Add_{Syn} are around 3:1 to 4:1. Introducing lock-and-unlock *after* the *CS* body is not rare.

Rem, **Rem_{All}-vs-Rem_{Syn}**: *CS* removals are about as often as additions. Different from Add_{Syn}, removing synchronization separately from the *CS* body (Rem_{Syn}) is rare.

Mod, **Mod_{Body}-vs-Mod_{Syn}**: Modifications happen more frequently to the body of a *CS* than to the lock-unlock synchronization. However, modifications to synchronization are non-negligible, contributing to 38% of all modifications and involving 26% of all *CS*es in our study.

Mod_{SynP}: Changing synchronization primitives are quite common in Apache, Mozilla, and MySQL. They are mainly due to three reasons: (1) functionality enhancement that allows lock profiling and deadlock monitoring, which happens in both MySQL (in mysql_mutex) and Mozilla (in AutoLock); (2) performance enhancement, such as replacing regular locks with reader-writer locks; (3) fixing bugs introduced by earlier primitive changes, which happens in MySQL; (4) readability enhancement through wrapper functions.

[4] As mentioned in Section 2, we will present Mozilla results with and without considering `AutoLock` separately in Table 3, Table 4, and Figure 1 – 3. For consistency, all the numbers presented in the text of this section consider basic locks only; all the qualitative observations presented here are true for both considering and not considering `AutoLock`. We will also discuss `AutoLock` changes at the end of this sub-section.

Table 3: Number of changes with certain pattern (Subscripts in Mozilla column are `AutoLock` numbers)

	Apache	Mozilla	MPlayer	MySQL
Add	138	227_{1809}	65	1548
Add_{All}	111	183_{1379}	48	1182
Add_{Syn}	27	44_{430}	17	366
Rem	199	272_{1088}	59	1411
Rem_{All}	199	272_{1086}	59	1395
Rem_{Syn}	0	0_{2}	0	16
Mod	467	204_{671}	33	4301
Mod_{Body}	291	$165_{n/a}$	23	2622
Mod_{Syn}	176	39_{671}	10	1679
Mod_{SynV}	17	6_{182}	3	117
Mod_{SynP}	109	6_{489}	0	816
Mod_{SynB}	38	$7_{n/a}$	4	577
Mod_{SynS}	0	$0_{n/a}$	0	28
Mod_{SynU}	12	$11_{n/a}$	0	141
# all changes	804	703_{3568}	157	7260

Table 4: Number of *CS*es w/ specific modifications (The subscripts in Mozilla column are `AutoLock` numbers)

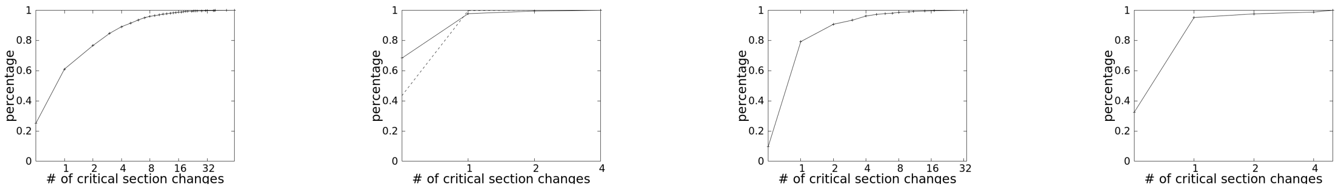
	Apache	Mozilla	MPlayer	MySQL
Mod	261	149_{500}	23	1173
Mod_{Body}	196	$139_{n/a}$	23	932
Mod_{Syn}	93	15_{500}	9	640
Mod_{SynV}	14	5_{234}	2	88
Mod_{SynP}	73	6_{410}	0	555
Mod_{SynB}	32	$5_{n/a}$	4	467
Mod_{SynS}	0	$0_{n/a}$	0	28
Mod_{SynU}	12	$11_{n/a}$	0	141
# all *CS*es	366	357_{2541}	76	2095

How Many Changes for Each *CS*? Change is the norm. As shown by Figure 1, only about 10% of *CS*es have encountered no changes after being added to Apache. This ratio is around 30% for MySQL and MPlayer, and higher for Mozilla, likely because Mozilla has the youngest code repository.

The majority of *CS*es that have been changed (modified or removed) are changed for 1 – 4 times throughout the revision history, as shown in Figure 1. Of course, highly changed *CS*es do exist. MySQL and Apache each has more than 2% of *CS*es changed for more than 10 times. For example, a 668-line *CS* in MySQL was changed for 39 times in 6 years.

Statistical Correlation Test We use Spearman's rank correlation coefficient [54] to explore what features are most correlated with the number of changes to a *CS* c. We consider three sets of features: (1) features reflecting the property of c itself, including length and age; (2) features reflecting the file f holding c, including the number of revisions that involves f, the number of *CS*es inside f, and the total number of changes to other *CS*es in f; (3) features reflecting the lock v that protects c, including the total number of *CS*es protected by v, and the total number of *CS* changes to other *CS*es protected by v. Our data set includes all *CS*es protected by global locks[5] in the latest version of MySQL. Results show that, among all the features under comparison,

[5] We cannot accurately know which *CS*es share a heap lock.



(a) MySQL (b) Mozilla (c) Apache (d) MPlayer

Figure 1: Cumulative distribution of #changes encountered by each *CS* (*CS* additions are not counted; the dashed line also considers `AutoLock`)

Table 5: Purposes of *CS* changes

	Apache	Mozilla	MPlayer	MySQL	Tot.
Correctness	13	9	20	21	63
Functionality	13	22	12	10	57
Maintainability	16	9	17	11	53
Performance	6	9	1	1	17
Robustness	2	1	0	7	10
Total	50	50	50	50	200

Table 6: Purposes of changes w/ different patterns

	Robu.	Main.	Func.	Correct.	Perf.	Tot.
Add_{Syn}	0	0	0	30	0	30
Mod_{SynV}	0	5	5	5	15	30
Mod_{SynB}	0	4	0	13	3	20
Mod_{SynS}	0	0	2	2	6	10
Mod_{SynU}	0	2	5	13	0	20

the number of Mod_{Body} changes and the number of Mod_{SynB} changes to a *CS* c are most correlated with c's size; the number of changes of other patterns and the total number of all changes to c are most correlated with the number of total revisions involving file f. These are all statistically strong correlations.

`AutoLock` in Mozilla About half of the *CS*es in the latest version of Mozilla use `AutoLock`. Since `AutoLock` was introduced later than the basic lock in Mozilla, the code repository contains many more activities associated with it than basic lock, as shown in Table 3 and 4. As we can see from the tables and figures above, the observations discussed above apply to both basic-lock *CS*es and `AutoLock` *CS*es. Specifically, for `AutoLock` *CS* changes, the ratio between Add_{All} and Add_{Syn} is about 3:1; *CS* removals are common, as well as *CS* additions; synchronization changes are common for `AutoLock` *CS*es. These are all similar with non-`AutoLock` *CS* changes. In comparison, an even larger portion of `AutoLock` *CS*es went through changes in Mozilla. Due to the similarity between the change patterns of basic *CS*es and `AutoLock` *CS*es, we will not separately discuss them for the remainder of this paper.

3.1.2 Why Did Changes Happen?

To understand change purposes, we manually investigate randomly sampled changes.

General Purposes Table 5 shows the purpose breakdown of 200 randomly sampled changes, with 50 from each software project. As we can see, correctness fixes, functionality changes, and code refactoring are almost equally common, each leading to about 25% of all changes. Performance enhancement is not negligible, leading to 8.5% of changes.

Pattern-Specific Purposes Due to space constraints, we only discuss the purpose break-downs of five structural patterns in Table 6 and below. We sampled 30 cases for patterns that have more than 400 changes (Add_{Syn} and Mod_{SynV}), 10 cases for patterns that have fewer than 100 total changes (Mod_{SynS}), and 20 for the other two patterns.

$\mathbf{Add}_{\mathbf{Syn}}$: When lock-and-unlock synchronization is added around a code region, it is always for avoiding concurrency bugs. To understand why the lock-and-unlock was not added earlier, we further studied the code change history for 30 cases. In 6 cases, synchronization was not needed when the code region was first introduced, but was demanded later due to software changes. In all other cases, not adding lock-and-unlock together with the *CS* body is buggy.

$\mathbf{Mod}_{\mathbf{SynV}}$: Interestingly, more changes are made for performance reasons, where fine granularity locks replace coarse granularity locks, than for correctness reasons.

$\mathbf{Mod}_{\mathbf{SynB}}$: Boundary adjustments are quite common, affecting almost 20% of all *CS*es. They are mainly used for fixing concurrency bugs, where code regions right outside a *CS* should be moved inside to avoid atomicity violations.

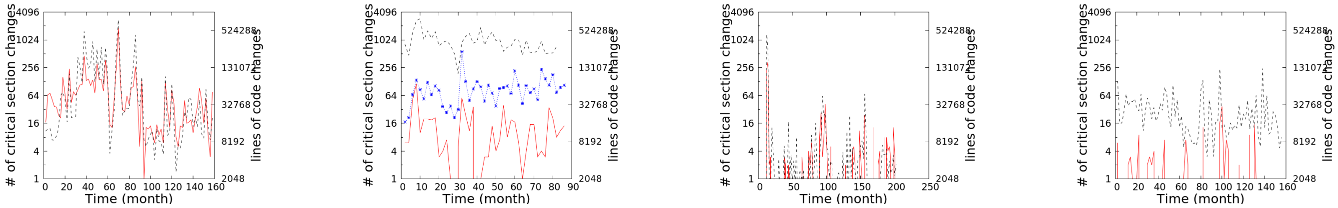
$\mathbf{Mod}_{\mathbf{SynS}}$: Splits are not common, maybe because they are complicated to reason about, which will be discussed more in Section 4. In 6 out of 10 examined cases, splits are conducted to avoid blocking competing threads for too long and hence to improve performance. In one case, the *CS* is split to fix a deadlock bug; in another case, the split moves part of the *CS* earlier to fix an order violation bug.

$\mathbf{Mod}_{\mathbf{SynU}}$: Adding unlocks have happened to 5.6% of *CS*es. More than half of these are simply because developers forgot to release a lock, particularly on error-handling paths right before function returns.

3.1.3 When Did Changes Happen?

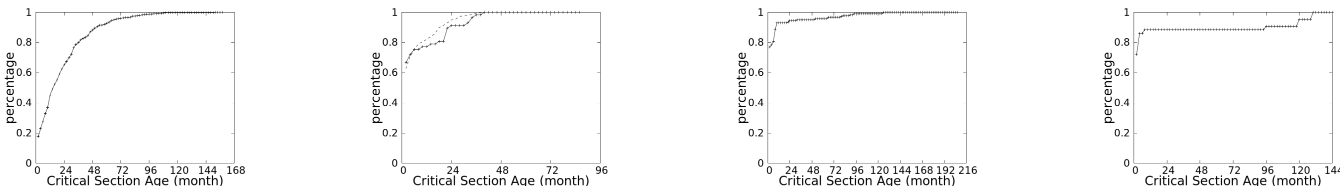
Regarding Software Age As we can see in Figure 2, the number of changes is relatively stable over the time, not getting significantly more or less with software getting older. Probably not surprisingly, the number of *CS* changes and the lines of changed code roughly follow the same trend over time, as shown in Figure 2. Based on Spearman's rank correlation coefficient and Z test, these two sequences indeed have *strong correlation* in all four projects, with 95% statistical confidence.

Regarding *CS* Age As we can see in Figure 3, the change frequency of a *CS* does drop with the *CS* getting older. 60–80% of changes to a *CS* happen within the



(a) MySQL (b) Mozilla (c) Apache (d) MPlayer

Figure 2: Number of *CS* changes, shown by solid red lines, and lines of changed code, shown by dashed lines, over software ages. (We compute software age by counting how long the software has lived since its first publicly released version; the dotted blue line in Mozilla figure also considers `AutoLock`.)



(a) MySQL (b) Mozilla (c) Apache (d) MPlayer

Figure 3: Percentage of cumulative changes over the age of *CS*es (*CS* additions not counted; the dashed curve in Mozilla also considers `AutoLock`.)

first two months of the *CS* birth in Mozilla, Apache, and MPlayer. Among the changes that happen after two years of a *CS*'s birth, the most common pattern is *CS* body modification, followed by *CS* removals.

Performance vs. Correctness One might wonder whether performance-enhancement changes tend to happen at older ages of a *CS* than correctness bug fixing changes. After checking 20 changes sampled from each type, t-test shows that the difference between these two is *not* statistically significant. In fact, the average *CS* ages for both types of changes are around 4 years, much larger that of all changes, which is around 1 year.

3.2 Discussion

To Lock or Not to Lock? That is a difficult question. For as many as 20–25% of *CS* additions in the studied projects, lock-and-unlocks are added *after* *CS* bodies. According to our manual check, many of these *CS* bodies should have been protected from the very beginning (24 out of 30 sampled cases), while developers' ignorance of synchronization needs caused concurrency bugs. In other cases where locks were not needed at the beginning (6 out of 30 cases), developers had to track software changes and add extra synchronization to previously implemented code regions. Making things more complicated, *CS* removals are almost as common as additions. This further burdens developers with synchronization decisions.

This part of study further motivates concurrency-bug avoidance and detection research — it is very common for developers to forget locks when they implement *new* code. It also calls for tool support that can analyze software changes and infer changing synchronization needs on *old* code regions, which will guide developers to add or remove locks in old code regions. This has not been well studied in the past.

To (Un)Lock Here or There? To Lock *A* or *B*? These are difficult questions. Our study shows that, after recognizing the need to synchronize, deciding the details of synchronization is difficult. Adjusting lock and unlock locations (Mod_{SynS}, Mod_{SynB}, Mod_{SynU}), lock variables (Mod_{SynV}), and lock primitives (Mod_{SynP}) of existing *CS*es are common tasks for developers. Altogether, they contribute to more than 40% of all *CS* modifications and affect about a quarter of all *CS*es (Mod_{Syn} in Table 3 and 4).

This part of study calls for tool support to decide synchronization details — a tool that can automatically adjust the boundaries and lock variables of *existing* *CS*es with both correctness and performance in mind. This is *different* from generic concurrency-bug detection or fixing tools. Specifically, synthesizing all the synchronization in a large software project from scratch is probably unfeasible. However, automating the adjustment process is not only feasible, but also very helpful. It can leverage the common adjustment patterns taken by developers, as well as previous work on concurrency-bug detection, fixing, and lock-insertion [4, 24, 34, 61]. It will relieve developers' burden of synchronization changes (i.e., Mod_{SynB}, Mod_{SynP}, Mod_{SynS}, Mod_{SynU}, Mod_{SynV}).

How Common Are Correctness and Performance Problems? Correctness is behind a significant portion of all changes (31%). Performance changes are also not rare (8.5%), and are a big part of synchronization modifications and removals, as shown in Table 6. Furthermore, the real problem could be more than what reflected by changes, because developers may not realize problems in their software. This results further indicate that research support for both synchronization correctness and synchronization performance is needed. More studies along these two directions will be presented in Section 4 and 5.

Are Synchronization Problems Getting Harder or Easier with Software Evolving? Our study shows that synchronization problems are a common theme for an evolving software. Inevitably, functionality changes, robustness enhancement, and others would happen inside existing *CS*es. Even when existing *CS*es become stable, new *CS*es are introduced with software evolving. Fortunately, the four projects under study have shown no sign of synchronization problems getting worse with software getting larger/older.

How about Other Types of Synchronization? Apart from locks, condition variables are the second most popular synchronization primitive in C/C++ programs. We studied all changes to condition-variable `signal`/`wait` operations in these four software projects. We briefly discuss main observations below.

First, `signal`/`wait` changes are common, especially considering the number of `signal`s and `wait`s in each code version. For example, the latest version of MySQL contains 271 `signal`/`wait` operations, fewer than $\frac{1}{10}$th of lock/unlock operations, while the code repository contains 1484 changes to `signal`s and `wait`s.

Second, many changes are made to adjust existing `signal`–`wait` pairs. Across these four projects, 20 – 60% of `signal` (or `wait`) changes are made without accompanying `wait` (or `signal`) changes. We consider a `signal` change and a `wait` change to accompany each other, if they are from the same revision and use the same condition variable.

Third, a big portion of changes are made due to correctness issues. Among the 40 randomly sampled cases, about 40% of them are made for avoiding concurrency bugs.

Overall, developers have to frequently adjust synchronization details, such as where to `signal` for given `wait`s or where to `wait` for given `signal`s, to avoid concurrency bugs. Tool support will be useful.

Summary The study above shows that good tools are needed to help (1) judge whether there is a need for adding (for both newly written code and already existing code) or removing lock synchronization; (2) adjust synchronization details for performance and correctness concerns, which applies to both lock synchronization (i.e., adjusting *CS* boundaries and variables) and condition-variable synchronization (i.e., adjusting `signal`s and `wait`s); (3) tackling over-synchronization issues; and others.

4. OVER SYNCHRONIZATION STUDY

Over-synchronization happens when unnecessary synchronization is added to the software. It would overly constrain software interleaving and lead to performance degradation.

Although many empirical studies have looked at real-world concurrency bugs [9, 14, 29, 45, 70], almost *no* study has focuses on how developers handle over-synchronization problems in real world. It will be the focus of this section.

Table 7: 20 over-synchronization related changes

	Apache	Mozilla	MPlayer	MySQL
Mod_{SynV}	3	2	0	6
Mod_{SynB}	0	0	0	3
Mod_{SynS}	-	-	-	6
Total	3	2	0	15

4.1 Methodology and Threats to Validity

Following the study in Section 3.1.2, we focus on three types of *CS* changes with dense population of over-synchronization issues — Mod_{SynV}, Mod_{SynS}, and Mod_{SynB}.

```
  lock(&LOCK_thread_count);
  while ((tmp=it++)) {
    if (...) {
-     expensive_operation(tmp);
+     lock(&tmp->LOCK_delete);
      break;
    }
  }
  unlock(&LOCK_thread_count);
+ if (...) {
+   expensive_operation(tmp);
+   unlock(&tmp->LOCK_delete);
+ }
```

Figure 4: A *CS* split from $\text{MySQL}_{\text{r1233}}$

Among the cases in Table 6, these three patterns provide 20 changes that fix/relieve over synchronization issues, as shown in Table 7.

Note that over-synchronization goes far beyond these 20 cases, which are collected from 30, 20, and 10 randomly sampled Mod_{SynV}, Mod_{SynS}, and Mod_{SynB} changes. They do not cover all over-synchronization issues fixed by these three types of changes, not to mention over-synchronization issues fixed by other types of changes. There are also over-synchronizations not fixed yet, which might be many given the preliminary support for over-synchronization detection and fixing. These 20 cases serve as a starting point of understanding real-world over-synchronization issues.

4.2 Observations

Where to Apply the Changes Naturally, these three types of over-synchronization fixes are typically applied to *CS*es with highly contended locks and time-consuming operations, such as system calls and the processing of big data-structures. For example, three MySQL split cases relieve the contention on LOCK_thread_count, which is a hot lock used by more than 10 *CS*es, including four sections invoked during every iteration of busy loops. As another example, $\text{Mozilla}_{r166150}$ helps relieve the contention on globalMutex, a default lock shared by *CS*es in Mozilla-ICU component.

How to Conduct Mod_{SynS} Conceptually, a *split* cuts a *CS* C protected by lock L into at least two parts, C_1 and C_2, each protected by a lock. Conducting a split involves several challenges: (1) how to protect the split-out code; (2) how to protect the newly created gap between C_1 and C_2; (3) how to re-structure the code to complete the split.

For the first issue, in about half of the cases, every split is still protected by the original lock, such as that shown in Figure 5[6]; in the other cases, the split-out code is protected by a different lock that is more specialized, such as the per-object tmp→LOCK_delete replacing the original global lock LOCK_thread_count shown in Figure 4.

For the second issue, in about half of the cases, C_1 and C_2 do not need to be put inside one *CS*. They were put together due to code-structure/readability benefits, as shown in Figure 4. In other cases, C_1 and C_2 were intended to be atomic together. The developers had to play some tricks, such as making the lock protecting C_2, which is different from L, also protect part of C_1 (MySQL_{r4591}), or copying the values of some shared variables used by both C_1 and C_2 into local variables (e.g., ncell in Figure 5). Sometimes, developers made semantic sacrifices to enable the split. For example,

[6] Throughout the paper, '+' denotes lines added by a revision; '-' denotes lines deleted by a revision; the code shown in figures is simplified for demonstration purpose; all comments are added by authors.

```
  lock(&btr_search_latch);
+ ncell = hash_get_n_cells(hash_index);
+ for (i=0; i<ncell;                            i++) {
- for (i=0; i<hash_get_n_cells(hash_index); i++) {
+   if ((i!=0)&&((i%CHUNK_SIZE)==0)) {
+     unlock(&btr_search_latch);
+     os_thread_yield();
+     lock(&btr_search_latch);
+   }
    ...
  }
  unlock(&btr_search_latch);
```

Figure 5: A *CS* split from $\mathbf{MySQL_{r2121}}$

after the *CS* split in MySQL_{r4591}, SHOW GLOBAL STATUS can no longer guarantee to aggregate the status information of all active threads.

The third issue is surprisingly tricky. Naively, if a *CS* only contains straight-line code, it can be split by simply inserting an unlock and a lock. The reality is more complicated due to control flows, particularly loops, surrounding the *CS*, as demonstrated in Figure 4.

How to Conduct $\mathbf{Mod_{SynV}}$ Changing lock variables mainly involve two challenges: (1) selecting a new lock; (2) finding all *CS*es that need lock-variable replacement.

Surprisingly, in all 11 cases under study, the original lock is replaced by a brand-new lock, newly declared and introduced into the software in the corresponding revision. In about half the cases, the newly introduced lock is only used to protect one static code region, making this region safe to be executed by multiple threads in parallel. In five cases, a global lock is replaced by per-object locks. In other cases, global locks are replaced by more specialized global locks.

Finding all *CS*es that need lock-variable replacement is an error prone process. Although not related to over-synchronization, Apache HTTPD once tried to rename a lock from proxy_module−>mutex to proxy_mutex. Developers kept missing *CS*es and took four revisions to finally finish all the needed replacement, which introduced bugs.

How to Conduct $\mathbf{Mod_{SynB}}$ The key challenge in boundary change is to identify a code region near the boundary of a *CS* that can be moved out without introducing concurrency bugs or damaging data dependency. Sometimes, this reasoning is easy. For example, MySQL_{r300} moves a condition-variable broadcast out of a *CS*. Sometimes, this requires more program semantics knowledge. For example, in MySQL_{r152}, developers realize that their code only reads one log entry, instead of multiple, and hence can be conducted outside the *CS*.

4.3 Discussion

Over-synchronization is a real problem, and is cared by developers. Developers change synchronization primitives to enable lock-contention profiling in MySQL and Mozilla, and sometimes relieve over synchronization at the cost of code readability or functionality (Figure 4 and 5).

Our study demonstrates that discovering and fixing over-synchronization take a lot of manual effort and are error prone. (1) All three types of changes/fixes discussed above can potentially introduce concurrency bugs and demand non-trivial synchronization correctness reasoning. (2) Many new lock variables are introduced during these fixes (Mod_{SynS} and Mod_{SynV}). The ad-hoc way of introducing these variables can easily lead to correctness and/or maintenance problems. (3) The code movement during these fixes is often non-trivial and could break single-thread semantics (Mod_{SynS} and Mod_{SynB}).

Our study also shows that it is feasible to develop tools to automate part of the over-synchronization detection and fixing process. Some common patterns of Mod_{SynS}, Mod_{SynV}, and Mod_{SynB} discussed above can help build such tools. A large part of over-synchronization reasoning is about synchronization correctness, which is shared by previous research on concurrency bugs. Language and run-time techniques [4, 24, 34, 58] may also help transparently address some of these issues.

5. CONCURRENCY BUG ORIGINS

5.1 Methodology and Threats to Validity

Software and Bug Sources Concurrency bugs used by this study come from two sources. The first includes *all* 28 real-world concurrency bugs, coming from more than ten widely used C/C++ software projects, repeated and evaluated in four recent concurrency-bug papers [20, 53, 70, 71]. This is our main source, because (1) the root cause of these bugs are well understood, which allows us to accurately identify their origins; and (2) these bugs have been widely used as benchmarks in state-of-the-art concurrency-bug literature [2, 15, 21, 29, 30, 32, 42, 45, 60, 61, 63, 66]. In our work, for each bug, we manually checked the user-reported buggy version and all the related previous versions to identify at which version the bug was introduced.

The second source is our critical-section change study. We check the origins of 12 randomly sampled concurrency bugs whose root causes are described in the revision log. We did not use more cases from this source, because revision log often does not describe bug root causes in detail and hence is not a good source for our in-depth bug-origin study.

Taxonomy Our categorization is based on three key ingredients of a concurrency bug: (1) shared variable(s); (2) instructions accessing these shared variables; and (3) synchronization contexts, such as surrounding locks and preceding barriers, that fail to enforce correct ordering among these instructions. The code revision that introduces a concurrency bug, referred to as *buggy revision*, must bring some or all of these ingredients into the software. Our categorization is based on which ingredients are introduced.

Study Mechanisms None of the bugs studied here have their origins mentioned in the bug reports or revision logs. For each bug, we first understand its three key ingredients and then manually search through the corresponding code repository for the first version that contains all ingredients.

Threats to Validity Due to difficulty of identifying bug origins, we choose to focus on bugs that have been repeated and hence can be thoroughly understood, in order to provide accurate results. The trade-off is that there are not many real-world C/C++ concurrency bugs that have been repeated and discussed in research literature. We checked *all* real-world C/C++ concurrency bugs used by a set of recent work [20, 53, 70, 71] without any bias. We also complement the above bug suite with randomly sampled bugs whose fixes are mentioned in the revision logs. One possible and uncontrollable bias is that bugs with complicated root causes may be less likely to get repeated by previous research or discussed in logs.

Having said that, we believe our study is a necessary step in understanding concurrency-bug origins. Our suite of

Table 8: How concurrency bugs are introduced (The subscripts represent b(ug) ids or r(evision) ids. The superscripts, A/O/D/$\mathbf{A}_m$, represent common root-cause patterns [29]: single-variable atomicity violations, order violations, deadlocks, and multi-variable atomicity violations. Td represents thread.)

		New Variable	New Instruction Td 1	New Instruction Td 2	New Context Td 1	New Context Td 2
Type 1	Aget$^{\mathbf{A_m}}$, Apache$^{\mathbf{D}}_{b42031}$, Mozilla$^{\mathbf{D}}_{r996770}$ MySQL$^{\mathbf{A}}_{b791}$, MySQL$^{\mathbf{A_m}}_{r1810.2246.1}$, MySQL$^{\mathbf{D}}_{r1110.10.2}$	-	✓	-	-	-
Type 2	Apache$^{\mathbf{A}}_{b25520}$, Click$^{\mathbf{O}}$, MySQL$^{\mathbf{A}}_{b3596}$	-	✓	-	✓	-
Type 3	HTTrack$^{\mathbf{O}}_{b20247}$ Mozilla$^{\mathbf{D}}_{b79054}$, Mozilla$^{\mathbf{A/O}}_{b142651}$, Mozilla$^{\mathbf{D}}_{b679524}$ MPlayer$^{\mathbf{D}}_{r30851}$ MySQL$^{\mathbf{A}}_{r703}$, SQLite$^{\mathbf{D}}_{b1672}$, Transmission$^{\mathbf{O}}_{b1818}$, x264$^{\mathbf{O}}$	✓	✓	✓	-	-
Type 4	Apache$^{\mathbf{D}}_{r88671}$, Apache$^{\mathbf{A}}_{r103588}$, Apache$^{\mathbf{A}}_{r1201146}$, Cherokee$^{\mathbf{A}}_{b326}$ Mozilla$^{\mathbf{O}}_{b61369}$, MySQL$^{\mathbf{A_m/O}}_{b2011}$, ZSNES$^{\mathbf{O}}_{b10918}$	✓	*	✓	✓	✓

```
/* Log Thread */

/*log status was OPEN*/
  ... //close old log
+ log_status = CLOSED;
  ... //open new log
  log_status = OPEN;
```

```
/* Query Thread */
/*log after transaction*/
if(log_status==OPEN){
  ... // log update (#)
}else{
  //Failure:transaction
  // not logged
}
```

Figure 6: How the MySQL_{791} bug was introduced

```
                + static httrackp *g_opt = NULL;

/*    Main Thread   */              /*    Child Thread   */
 int main() {                         void child(...) {
   ...                                  ...
   pthread_create(child, ...);          /*Initialize g_opt*/
+  mutexlock(&g_opt->s.l);         +    g_opt = create_opt();
   ...                                  ...
 }                                    }
```

Figure 7: How HTTrack_{b20247} was introduced

bugs come from a representative set of open-source multi-threaded C/C++ software, cover a wide variety of root-cause patterns and failure patterns, and have been widely used in the research community. The two different sources of bugs in our study end up showing consistent trends of origins, as shown below.

5.2 Observations

Among the 40 bugs that we studied (28 from previous papers and 12 from revision logs), 15 have unknown origins, as they exist in the first publicly available version of their respective projects. The remaining 25 bugs are introduced in four different ways, as shown in Table 8. In the discussion below, we will call the items introduced by the buggy revision as *new* and the ones that exist prior to the buggy revision as *old*.

Type 1: One Thread, New Instructions in Old Contexts A concurrency bug is introduced by changes in a single thread, where some new memory instructions are inserted in old synchronization contexts. This happens to both atomicity violation bugs and deadlocks in our study.

For example, Figure 6 explains how a single-variable atomicity violation was introduced in MySQL. In one revision, developers decide to update log_status to be CLOSED, shown by '+' in Figure 6. This change makes perfect sense for the semantics of the logging thread. However, it could cause the query thread to skip transaction logging, a severe security vulnerability, if the query thread reads log_status after it is set to CLOSED and before it is set back to OPEN. Note that, the old version had no logging-related problems, because the logging code in the query thread, denoted by # in Figure 6, can handle temporarily unavailable logs.

Type 2: One Thread, New Instructions in New Contexts The buggy revision introduces a new code region with a new synchronization context, which is not well synchronized with some old code in another thread. We observe both atomicity violations and order violations in this category. Specifically, the buggy revisions of Click, x264, and MySQL_{b3596}, all create new threads that do not synchronize well with old threads. Click's new thread could read shared variables after they are destroyed by old threads; x264's new thread could read shared variables before they are initialized by old threads; MySQL_{b3596}'s new thread accesses shared variables without using the proper lock used by old threads.

Type 3: Multiple Threads, New Variables Accessed in Old Contexts There exists code regions r_1 and r_2 that can execute concurrently in the old version. The buggy revision introduces a new variable accessed by both regions. The lack of synchronization between these two regions leads to concurrency bugs. In $\text{Transmission}_{b1818}$ and HTTrack_{b20247}, concurrent accesses from two concurrent regions cause a new shared variable to be read before initialization (i.e., order violations); in Apache_{b25520}, $\text{Mozilla}_{b142651}$, and MySQL_{r703}, the concurrent read-and-write accesses from two concurrent regions lead to atomicity violations. In several other cases, two threads can request lock A concurrently in the old version. Inserting lock-B acquisitions to be before and after the lock-A acquisitions in these two threads causes deadlocks.

Type 4: Multiple Threads, New Instructions in New Context(s) This typically happens when the revision introduces new multi-threaded components into the software or significant re-implementation for many threads (e.g., Cherokee_{b326}). The shared variables involved in these bugs are mostly new variables, except for $\text{Apache}_{r1201146}$.

5.3 Discussion

Facing large real-world multi-threaded software, it is critical to improve the performance and accuracy of existing concurrency-bug analysis techniques. This could be helped through history/change awareness [18, 57, 67], an approach that has **not** been well explored. Our study shows how this approach can help two critical and time-consuming components of concurrency-bug analysis: analyzing which instructions access same variables (i.e., *memory-access analysis*), and analyzing what are the synchronizations around these instructions (i.e., *synchronization analysis*).

First, synchronization analysis can be significantly simplified for many bugs through history awareness. With old synchronization-context information, about half of the studied bugs would require **no** new synchronization analysis to be detected, because their buggy code is inside completely old synchronization contexts. Furthermore, another 20% of the studied bugs involve old synchronization context in one thread, and hence could also benefit from history awareness.

Second, memory-access analysis can be significantly simplified for many bugs through history awareness. About half of the studied bugs only involve new variables accessed by new instructions with pointers propagated through new instructions. Therefore, detecting them only requires memory-access analysis for the changed code, instead of the whole program. This simplification is huge, as the size of a revision is often less than 0.01% of the whole software. For the remaining bugs, detecting them can leverage incremental pointer-alias analysis [31, 51, 59, 69], as they involve old memory instructions or old shared variables or both.

Third, about a quarter of the studied bugs can benefit from both *almost-no* synchronization analysis and *revision-local* memory-access analysis discussed above, and hence would require extremely simple analysis to discover. Figure 7 illustrates such a case for HTTrack_{b20247}. The revision introduces a new global pointer variable `g_opt`, which is initialized by the child thread and dereferenced by the main thread. The code regions of the initialization and the dereference have been concurrent since the old version. With history/revision-aware analysis, we can easily tell that the dereference of `g_opt` could happen before its initialization.

The above features can help improve not only analysis performance, but also analysis accuracy, as some saved analysis time can be used for improving accuracy. Furthermore, knowledge about false positives in analyzing old versions can help prune false positives in new versions.

The above features can help not only concurrency bug detection and testing, but also prevention. Since half of the examined bugs happen in code regions that can execute concurrently with each other in the old version, an IDE that highlights concurrent code regions could help prevent many bugs. Furthermore, lightweight history/change-aware analysis can provide developers instant feedback about concurrency bugs introduced by revisions.

Of course, there are also challenges. For example, reusing synchronization information (e.g., time-stamps and locksets) from old versions requires extra storage. It is also difficult to judge whether concurrency bugs are introduced based on the revision alone, as about three quarters of the studied bugs involve synchronization contexts or memory accesses inherited from old versions.

Overall, our origin study of concurrency bugs coming from different sources has delivered a consistent message — the awareness of history can help improve the performance and accuracy of many concurrency-bug related techniques.

6. RELATED WORK

Many characteristics studies have been conducted to understand general software bugs [6, 16, 41, 55]. Recently, studies also looked at concurrency bugs [9, 14, 29, 70, 71] and evaluate new synchronization primitives [48, 49, 60] based on bug databases and student/researcher experiences. Our study complements them by checking software code repositories, which reveals real-world code development information unavailable in bug databases.

A recent study checks performance bugs in bug databases [19]. It found 6 synchronization related bug reports among all its sampled performance bugs, with no details about these 6 cases. *None* of the 20 over-synchronization fix changes discussed in Section 4 can be found from bug databases.

Many studies have looked at software code repositories in the past [23, 25, 28, 40, 46, 50, 65], most of which did not focus on synchronization issues. Some recent studies look at parallel programs written using Java concurrent programing constructs [44], MPI [33], and C# Task Parallel Library [40]. The study by Xin et al. [62] looks at the frequencies of some lock usage patterns over three versions of four software projects, such as using lock after an if check and checking the return value of a lock acquisition. The study by Sadowski et al. [50] looks at how data races evolve over time in two Java programs. Specifically, they made two findings: (1) the number of racy variables remains high over time; (2) variables may go in and out of being racy over the course of a project. As we can see, although all looking at multi-threaded software, our study has different goals from previous studies. Consequently, our study collects different types of software change information and answers different types of questions, including over-synchronization issues and concurrency-bug origin issues, from previous work.

Many concurrency bug detection tools have been proposed [11–13, 21, 32, 36, 37, 52, 66, 68, 70]. Almost all of them focus on one software version at a time, and hence can benefit from our study of concurrency bug origins.

Tools have been built recently to detect performance bugs [17, 22, 38, 39, 64], detect false sharings [27, 43], and profile locks [8, 56]. Our study provides motivation and guidance for future research to tackle over-synchronization issues.

New constructs have been designed to (partly) replace locks and ease synchronization [1, 3, 24, 58]. Although locks are still the most commonly used synchronization constructs in open-source C programs [40], our study shows that developers indeed face challenges of using locks.

Our concurrency-bug origin study is inspired by traditional revision impact analysis designed for sequential bugs [5, 47]. Previous results cannot be applied to concurrency-bug research, because concurrency bugs involve multiple threads and synchronization.

7. CONCLUSION

This paper studies code repositories to understand synchronization challenges encountered by real-world developers. We first check over 250,000 code revisions in the code repositories of four representative C/C++ software projects to figure out how many critical section related changes are there, why the changes are made, and when they are made. We then conduct thorough case studies to better understand how concurrency bugs are introduced by code changes and how developers handle over-synchronization problems. Our findings provide insights and motivation for future research on tackling synchronization problems, both lack-of-synchronization and over-synchronization problems.

8. REFERENCES

[1] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. Sharc: Checking data sharing strategies for multithreaded c. In *PLDI*, 2008.

[2] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[3] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[4] Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.

[5] Ophelia C. Chesley, Xiaoxia Ren, Barbara G. Ryder, and Frank Tip. Crisp–a fault localization tool for java programs. In *ICSE*, 2007.

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88, 2001.

[7] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*, 2013.

[8] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*, 2014.

[9] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS*, 2003.

[10] Daniel Fischer. Getting started with bazaar for mysql code. http://dev.mysql.com/tech-resources/articles/getting-started-with-bazaar-for-mysql.html.

[11] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[12] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

[13] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[14] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN*, 2010.

[15] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndstrike: Toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, 2011.

[16] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen C. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*, 2009.

[17] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, 2012.

[18] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *ISSTA*, 2011.

[19] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.

[20] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[21] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *ASPLOS*, 2012.

[22] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *FSE*, 2010.

[23] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *FSE*, 2012.

[24] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[25] Paul Luo Li, Ryan Kivett, Zhiyuan Zhan, Sung-eok Jeon, Nachiappan Nagappan, Brendan Murphy, and Andrew J. Ko. Characterizing the differences between pre- and post- release versions of software. In *ICSE*, 2011.

[26] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID*, 2006.

[27] Tongping Liu and Emery D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *OOPSLA*, 2011.

[28] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *FAST*, 2013.

[29] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[30] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[31] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *CC*, 2013.

[32] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[33] Cristina Marinescu. An empirical investigation on mpi open source applications. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 20:1–20:4, New York, NY, USA, 2014. ACM.

[34] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL*, 2006.

[35] Mozilla Developer Network. Getting mozilla source code using mercurial. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Source_Code/Mercurial.

[36] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Grard Basler, Piramanayagam A Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

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

[38] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

[39] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*, 2013.

[40] Semih Okur and Danny Dig. How do developers use parallel libraries? In *FSE*, 2012.

[41] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in linux: ten years later. In *ASPLOS*, 2011.

[42] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their finding places. In *ASPLOS*, 2009.

[43] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys*, 2010.

[44] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. A large-scale study on the usage of java's concurrent programming constructs. *J. Syst. Softw.*, 106(C):59–81, August 2015.

[45] Shanxiang Qi, Abdullah Muzahid, Wonsun Ahn, and Josep Torrellas. Dynamically detecting and tolerating if-condition data races. In *HPCA*, 2014.

[46] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar T. Devanbu. A large scale study of programming languages and code quality in github. In *FSE*, 2014.

[47] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *OOPSLA*, 2004.

[48] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *WDDD*, 2009.

[49] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using gcc and memcached. In *ASPLOS*, 2014.

[50] Caitlin Sadowski, Jaeheon Yi, and Sunghun Kim. The evolution of data races. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012.

[51] Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP*, 2005.

[52] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[53] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do i use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[54] Charles Spearman. Spearman's rank correlation coefficient. http://en.wikipedia.org/wiki/Spearman's_rank_correlation_coefficient.

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

[56] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP*, 2010.

[57] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. Recontest: Effective regression testing of concurrent programs. In *ICSE*, 2015.

[58] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[59] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *PLDI*, 2001.

[60] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. In *ASPLOS*, 2012.

[61] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In *OOPSLA*, 2011.

[62] Rui Xin, Zhengwei Qi, Shiqiu Huang, Chengcheng Xiang, Yudi Zheng, Yin Wang, and Haibing Guan. An automation-assisted empirical study on lock usage for concurrent programs. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 100–109, Sept 2013.

[63] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[64] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *ICSE*, 2012.

[65] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. How do fixes become bugs? In *FSE*, 2011.

[66] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA*, 2012.

[67] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. Simrt: An automated framework to support regression testing for data races. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 48–59, New York, NY, USA, 2014. ACM.

[68] Yuan Yu, Thomas Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[69] Jyh-shiarn Yur, Barbara G. Ryder, and William A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *ICSE*, 1999.

[70] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[71] Wei Zhang, Chong Sun, and Shan Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, 2010.