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.

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, 39, 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.

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 4:

- Over half of these bugs are introduced under old synchronization contexts (i.e., surrounding locks, preceding barriers/wait, 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. They are all large and mature projects under active development, with millions of lines of code in each version and 8 – 19 years of repository history. Our study looks at all the publicly available revisions from their version control systems — SVN for Apache and MPlayer, Mercurial for Mozilla, and Bazaar for MySQL. 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 CSes in the repository, each of which could live through many revisions; added-CS: CSes added since the first version; all-CS are more than added-CS, as some CSes exist in the initial code image; Section 2 discusses how we count CSes.)

<table>
<thead>
<tr>
<th>Application</th>
<th>Repository Info.</th>
<th>Latest Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Period</td>
<td>#Rev.</td>
</tr>
<tr>
<td>Apache HTTPD Web Server</td>
<td>1996 - 2014</td>
<td>25897</td>
</tr>
<tr>
<td>Mozilla Browser Suite</td>
<td>2007 - 2014</td>
<td>188000</td>
</tr>
<tr>
<td>MPayer Media Player</td>
<td>2001 - 2014</td>
<td>37000</td>
</tr>
<tr>
<td>MySQL Database Server</td>
<td>2000 - 2013</td>
<td>6800</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Structural Patterns</th>
<th>Purpose Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Fixing functional bugs</td>
</tr>
<tr>
<td>AddAll</td>
<td>Adding CSes</td>
</tr>
<tr>
<td>AddSyn</td>
<td>Synchronization and body added together</td>
</tr>
<tr>
<td>Rem</td>
<td>Removing CSes</td>
</tr>
<tr>
<td>RemAll</td>
<td>Synchronization and body removed together</td>
</tr>
<tr>
<td>RemSyn</td>
<td>Synchronization removed alone</td>
</tr>
<tr>
<td>Mod</td>
<td>Modifying existing CSes</td>
</tr>
<tr>
<td>ModBodyV</td>
<td>Critical section body modified</td>
</tr>
<tr>
<td>ModSyn</td>
<td>Critical section synchronization modified</td>
</tr>
<tr>
<td>ModSynV</td>
<td>Synchronization variable modified</td>
</tr>
<tr>
<td>ModSynP</td>
<td>Synchronization primitive modified</td>
</tr>
<tr>
<td>ModSync</td>
<td>Critical section boundary moved</td>
</tr>
<tr>
<td>ModSplit</td>
<td>Critical section split</td>
</tr>
<tr>
<td>ModUnlock</td>
<td>Adding unlock operations</td>
</tr>
</tbody>
</table>

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 CSes 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

```python
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 CSes 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 CSes are found, we discover a CS addition (Add) or removal (Rem), or an adding-unlock (ModSynU). If no CS is found, we discover a CS split (ModSplit), a CS removal (Rem), or an adding-unlock (ModSynU). Third, changes inside functions called by a lock-acquisition function are obtained by key-word search — lock, latch, and mutex — in the four software projects under study.

About half of Mozilla CSes 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 (ModBody); CS split (ModSplit), and adding unlock (ModUnlock) 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, CSes 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 (ModSyn). 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 CSes 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 CSes) does not affect our main observations. We could miss some ModSynV 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 (RemSyn) and CS-split (ModSyn), each contributes to at least 1.5% of all changes and affects at least 3.8% of all CSes, 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.

**Add: AddAll-vs-AddSyn:** Adding CSes contributes to 17–40% of CS changes in four projects. The ratio between AddAll and AddSyn are around 3:1 to 4:1. Introducing lock-and-unlock after the CS body is not rare.

**Rem: RemAll-vs-RemSyn:** CS removals are about as often as additions. Different from AddSyn, removing synchronization separately from the CS body (RemSyn) is rare.

**Mod: ModBody-vs-ModSyn:** 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 CSes in our study.

**ModSyn:** 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.

**How Many Changes for Each CS?** Change is the norm. As shown by Figure 1, only about 10% of CSes 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 CSes 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 CSes do exist. MySQL and Apache each has more than 2% of CSes 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 [3] 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 CSes inside $f$, and the total number of changes to other CSes in $f$; (3) features reflecting the lock $v$ that protects $c$, including the total number of CSes protected by $v$, and the total number of CS changes to other CSes protected by $v$. Our data set includes all CSes protected by global lock in the latest version of MySQL. Results show that, among all the features under comparison, 3

| Table 3: Number of changes with certain pattern (Subscripts in Mozilla column are AutoLock numbers) |
|-------------------------------------------------|-----|-----|-----|-----|
|                                   | Apache | Mozilla | MPlayer | MySQL |
| Add                                | 138   | 227,185 | 65   | 1548 |
| AddAll                             | 111   | 183,170 | 48   | 1182 |
| AddSyn                             | 27    | 44,310  | 17   | 366  |
| Rem                                | 199   | 272,108 | 59   | 1411 |
| RemAll                             | 199   | 272,106 | 59   | 1395 |
| RemSyn                             | 0     | 0      | 0    | 0    |
| Mod                                | 467   | 204,971 | 33   | 4301 |
| ModBody                            | 291   | 165,860 | 23   | 2622 |
| ModSyn                             | 176   | 39,071  | 10   | 1679 |
| ModSynV                            | 17    | 6,182   | 3    | 117  |
| ModSynP                            | 109   | 6,480   | 0    | 816  |
| ModSynH                            | 38    | 7,410   | 4    | 577  |
| ModSynS                            | 0     | 0,954   | 0    | 28   |
| ModSynU                            | 12    | 11,447  | 0    | 141  |
| # all changes                      | 804   | 703,158 | 157  | 7260 |

| Table 4: Number of CSes w/ specific modifications (The subscripts in Mozilla column are AutoLock numbers) |
|-------------------------------------------------|-----|-----|-----|-----|
|                                   | Apache | Mozilla | MPlayer | MySQL |
| Mod                                | 261   | 149,500 | 23   | 1173 |
| ModBody                            | 196   | 139,n/a  | 23   | 932  |
| ModSyn                             | 93    | 15,060   | 9    | 640  |
| ModSynV                            | 14    | 5,234    | 2    | 88   |
| ModSynP                            | 73    | 6,410    | 0    | 555  |
| ModSynH                            | 32    | 5,410    | 4    | 467  |
| ModSynS                            | 0     | 0,n/a    | 0    | 28   |
| ModSynU                            | 12    | 11,n/a   | 0    | 141  |
| # all CSes                         | 366   | 357,2541 | 76   | 2095 |

4As mentioned in Section 2, we will present Mozilla results with and without considering AutoLock separately in Table 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 subsection.

5We cannot accurately know which CSes share a heap lock.
AutoLock
dashed line also considers CS changes. In comparison, an even larger portion of CS AutoLock CS as and Add
ically, for AutoLock CS tables and figures above, the observations discussed above basic lock, as shown in Table 3 and 4. As we can see from the
itory contains many more activities associated with it than produced later than the basic lock in Mozilla, the code reposi-
3.1.2 Why Did Changes Happen?

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-

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 in-

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

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

### Table 5: Purposes of CS changes

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Apache</th>
<th>Mozilla</th>
<th>MPlayer</th>
<th>MySQL</th>
<th>Tot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>13</td>
<td>9</td>
<td>20</td>
<td>21</td>
<td>63</td>
</tr>
<tr>
<td>Functionality</td>
<td>13</td>
<td>22</td>
<td>12</td>
<td>10</td>
<td>57</td>
</tr>
<tr>
<td>Maintainability</td>
<td>16</td>
<td>9</td>
<td>17</td>
<td>11</td>
<td>53</td>
</tr>
<tr>
<td>Performance</td>
<td>6</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Robustness</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>200</td>
</tr>
</tbody>
</table>

### Table 6: Purposes of changes w/ different patterns

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AddSyB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>ModSyBV</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>15</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>ModSyB</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>13</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>ModSyS</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>ModSyAU</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>13</td>
<td>0</td>
<td>20</td>
</tr>
</tbody>
</table>

AddSyB: 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.

ModSyB: Boundary adjustments are quite common, affecting almost 20% of all CSes. They are mainly used for fixing concurrency bugs, where code regions right outside a CS should be moved inside to avoid atomicity violations.

ModSyS: 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.

ModSyAU: Adding unlocks have happened to 5.6% of CSes. More than half of these are simply because developers forgot to release a lock, particularly on error-handling paths right before function returns.
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 (ModSynB, ModSynU, ModSynV), lock variables (ModSynV), and lock primitives (ModSynP) of existing CSes are common tasks for developers. Altogether, they contribute to more than 40% of all CS modifications and affect about a quarter of all CSes (ModSyn 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 CSes 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][54][61]. It will relieve developers’ burden of synchronization changes (i.e., ModSynB, ModSynP, ModSynB, ModSynU, ModSynV).

**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, robust-
cess enhancement, and others would happen inside existing CSes. Even when existing CSes become stable, new CSes 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 `signals` and `waits` in each code version. For example, the latest version of MySQL contains 271 `signal`/`wait` operations, fewer than 1/3 of lock/unlock operations, while the code repository contains 1484 changes to `signals` and `waits`.

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 `waits` or where to `wait` for given `signals`, 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 `signals` and `waits`); (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] [11] [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.

### 4.1 Methodology and Threats to Validity

Following the study in Section 4.1.2, we focus on three types of CS changes with dense population of over-synchronization issues — ModSynV, ModSynB, and ModSynS.

### 4.2 Observations

**Where to Apply the Changes** Naturally, these three types of over-synchronization fixes are typically applied to CSes 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 CSes, including four sections invoked during every iteration of busy loops. As another example, Mozilla-166150 helps relieve the contention on `globalMutex`, a default lock shared by CSes in Mozilla-ICU component.

**How to Conduct ModSyns** Conceptually, a split cuts a CS C protected by lock L into at least two parts, C1 and C2, 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 C1 and C2; (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 4[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, C1 and C2 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, C1 and C2 were intended to be atomic together. The developers had to play some tricks, such as making the lock protecting C1, which is different from L, also protect part of C1 (MySQL_4590), or copying the values of some shared variables used by both C1 and C2 into local variables (e.g., `ncell` in Figure 5). Sometimes, developers made semantic sacrifices to enable the split. For example,
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 MySQLr1212

after the CS split in MySQLr4591, SHOW GLOBAL STATUS

The third issue is surprisingly tricky. Naively, if a CS only

The reality is more complicated due to

control flows, particularly loops, surrounding the CS, as

as demonstrated in Figure 4.

How to Conduct ModSynV Changing lock variables

mainly involve two challenges: (1) selecting a new lock; (2)

finding all CSes 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 intro-

duced outside the CS.

Finding all CSes 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. Develop-

ers kept missing CSes and took four revisions to finally finish

all the needed replacement, which introduced bugs.

How to Conduct ModSynB The key challenge in

boundary change is to identify a code region near the bound-

ary of a CS that can be moved out without introducing con-

currency bugs or damaging data dependency. Sometimes,

this reasoning is easy. For example, MySQLr309 moves a

condition-variable broadcast out of a CS. Sometimes, this

requires more program semantics knowledge. For example,

in MySQLr152, developers realize that their code only reads

one log entry, instead of multiple, and hence can be con-

ducted 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 er-

ror prone. (1) All three types of changes/fixes discussed

above can potentially introduce concurrency bugs and de-

mand non-trivial synchronization correctness reasoning. (2)

Many new lock variables are introduced during these fixes

(ModSynS and ModSynV). The ad-hoc way of introducing

these variables can easily lead to correctness and/or main-

tenance problems. (3) The code movement during these fixes

is often non-trivial and could break single-thread semantics

(ModSynS and ModSynB).

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 ModSynS, ModSynV,

and ModSynB discussed above can help build such tools. A

large part of over-synchronization reasoning is about syn-

chronization correctness, which is shared by previous re-

search on concurrency bugs. Language and run-time tech-

iques [1, 21, 33, 55] may also help transparently address

some of these issues.

5. CONCURRENCE 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 evalu-

ated in four recent concurrency-bug papers [20, 55, 70, 71].

This is our main source, because (1) the root cause of these

bugs are well understood, which allows us to accurately iden-

tify 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 in-

gredients of a concurrency bug: (1) shared variable(s); (2)

instructions accessing these shared variables; and (3) syn-

chronization contexts, such as surrounding locks and pre-

ceeding barriers, that fail to enforce correct ordering among

these instructions. The code revision that introduces a con-

currency bug, referred to as buggy revision, must bring some

or all of these ingredients into the software. Our categoriza-

tion 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 pro-

vide 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, 55, 70, 71] without any bias. We also com-

plement the above bug suite with randomly sampled bugs

whose fixes are mentioned in the revision logs. One possi-

ble 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(efision) ids. The superscripts, A/O/D/A_m, represent common root-cause patterns [29]: single-variable atomicity violations, order violations, deadlocks, and multi-variable atomicity violations. Td represents thread.)

<table>
<thead>
<tr>
<th>Type</th>
<th>New Instruction</th>
<th>New Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Aget^A_m, Apache^D_25247, Mozilla^D_110146</td>
<td>MySQL^A_791, MySQL^D_11810.2246.1, MySQL^D_1110.10.2</td>
</tr>
<tr>
<td>2</td>
<td>Apache^A_253120, Click^O, MySQL^A_63596</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>HTTrack^O_020247, Mozilla^D_1201146, Mozilla^A/O_1201146, Mozilla^D_16724, Transmission^D_1818 * x264^O</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>Apache^O_96671, Apache^A/O_103588, Apache^A_1201146, Cherokee^A_96326, Mozilla^O_61369, MySQL^D_2011, ZSNES^O</td>
<td>✓</td>
</tr>
</tbody>
</table>

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^D_13596, 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^D_13596’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 Transmission^O_020247 and HTTrack^O_020247, concurrent accesses from two concurrent regions cause a new shared variable to be read before initialization (i.e., order violations); in Apache^O_020247, Mozilla^A_1201146, and MySQL^D_703, 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^A_96326). The shared variables involved in these bugs are mostly new variables, except for Apache^O_1201146.

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 compo-
ments 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\textsubscript{2023.47}. The revision introduces a new global pointer variable \(g\text{opt}\), 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 \(g\text{opt}\) 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 \([3, 19, 41, 55]\). 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 \([23, 25, 28, 30, 40, 50, 55]\), most of which did not focus on synchronization issues. Some recent studies look at parallel programs written using Java concurrent programming 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. \([5\) 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 \([11, 19, 21, 32, 39, 51, 59, 63, 65, 70]\). 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 \([17, 22, 48, 59, 64]\), detect false sharings \([27, 43]\), and profile locks \([8, 54]\). 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 \([1, 3, 23, 58]\). 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 \([5, 17]\). 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


