How do Programmers Maintain Concurrent Code

Feiyue Yu and Hao Zhong
Department of Computer Science, Shanghai Jiao Tong University, China
Email: iamyfy@163.com, zhonghao@sjtu.edu.cn

***Abstract*—Concurrent programming is pervasive in nowadays software development. Although it unleashes the potential of multi-core computers, many programmers believe that concurrent programming is difficult, and maintaining concurrency code is error-prone. Although researchers have conducted empirical studies to understand concurrent programming, they still rarely study how programmers maintain concurrent code. To the best of our knowledge, only a recent study explored the modifications on critical sections, and many related questions are still open. In this paper, we conduct an empirical study to explore how programmers maintain concurrent code. In our empirical study, we analyze more concurrency-related commits and explore more issues such as the change patterns of maintaining concurrent code than the previous empirical study. In particular, according to our analysis on 696 concurrency-related commits, we summarize five change patterns. We apply our change patterns on the latest versions of three open source projects, and synthesize three pull requests. Until now, two of the pull requests have been accepted. Furthermore, we analyze other issues such as the usages of parallel API classes and the correlations between total commits and concurrency-related commits. Our results show that some of such usages follow specific trends and there is a correlation between total commits and concurrency-related commits. Such findings can be useful for programmers to maintain concurrent code and for researchers to implement treating techniques.**

I. Introduction

Many practitioners and researchers believe that the software maintenance phase is one of the most expensive phases, in the life cycle of a software system. Some reports (*e.g.*, [1]) claim that the software maintenance phase accounts for almost 80% of the whole budget. With the maintenance of software, many revision histories are accumulated [3]. Based on such revision histories, researchers have conducted various empirical studies to understand how programmers maintain code (*e.g.*, evolution of design patterns [2], fine-grained modifications [9], and the evolution of APIs [17]). These empirical studies deepen our understanding on software maintenance, and provide valuable insights on how to maintain code in future development.

In recent years, to fully leverage the potential of multi-core CPUs, concurrent programming has become increasingly popular [22]. For example, Pinto *et al.* [22] investigated 2,227 projects, and their results show that more than 75% of these projects employ some concurrency control mechanism. Despite of its popularity, many programmers find that concurrent programming is difficult [18], and often introduce relevant bugs in their code [15]. As it is difficult to maintain concurrent code, there is a strong need for a thorough empirical study on how programmers maintain such code. Despite of its importance, the topic is still rarely explored. To the best of our knowledge, only a recent study [10] was conducted to understand how programmers maintain concurrent code. Although the study is insightful and explores many aspects of concurrent programming, it is still incomplete. For example, their study sampled only 25 concurrency-related commits, and focuses on limited topics such over synchronization and how concurrency bugs origin. As a result, many relevant questions are still open. For example, are there any patterns, when programmers maintain concurrent code? Indeed, such patterns are useful for programmers when they maintain code. For example, Santos *et al.* [24] have explored the change patterns during software maintenance, and their results show that extracted change patterns can be applied to new code locations. However, their study does not touch the change patterns of concurrent code. A more detailed analysis can have the following benefits:

Benefit 1. The results can deepen the knowledge on how to maintain concurrent code. Due to the complexity of concurrent programming, we find that even experienced developers can be confused when they maintain relevant code. For example, Mark Thomas is a member of the Apache Tomcat Project Management Committee[1], and senior software engineer at the Covalent division of SpringSource[2]. He contributed more than 10,000 commits to Tomcat. In a commit message, he left the complaint as follow:

```
1  commit a6092d771ec50cf9aa434c75455b842f3ac6c628
2  Threading / initialisation issues. Not all were valid.
       Make them volatile anyway so FindBugs doesn't
      complain.
```

In this example, we find that even experienced programmers can have problems in understanding their own code changes, when they maintain concurrent code. Our results can resolve such confusions.

Benefit 2. The results can be useful to improve existing tools. For example, Meng *et al.* [19] proposed an approach that applies changes systematically based on a given example. With extensions, it can be feasible to apply our extract change patterns to update concurrent code. We further discuss this issue in Section IV.

However, to fulfill the above benefits, we have to overcome the following challenges:

Challenge 1. To ensure the reliability of our result, we have to collect many code changes that are related to concurrent programming. It is tedious to manually collect many related code changes for analysis. Tian *et al.* [26] worked on a similar research problem. They proposed an approach that identifies

[1]http://tomcat.apache.org/whoweare.html
[2]https://sourceforge.net/projects/covalent/

bug fixes from commits. Their results show that even advanced techniques can fail to identify many desirable commits.

Challenge 2. The changes on concurrent code can be complicated. A recent study [27] showed that only 38% commits are compilable. To analyze code that is not compilable, researchers typically use partial-code tools such as PPA [6] and ChangeDistiller [8] to analyze commits. However, as partial programs lose information, partial-code tools are imprecise and typically do not support advanced analysis. Furthermore, as we do not know what patterns can be followed, it is difficult to implement an automatic tool. As a result, it is inevitable to take much human effort when we conduct the empirical study.

In this paper, we conduct an empirical study on 98,325 commits that are collected from six popular open-source projects. To reduce the effort of manual inspection, we implement a set of tools that collect and identify concurrency-related commits automatically (see Section II-C for details). With its support, in total, we identified 11,868 concurrency-related commits, and manually analyzed 696 such commits. Based on our results, this paper makes the following contributions:

- The first analysis on the change patterns of maintaining concurrent programs. Based on our results, we summarize five change patterns, and we present their examples for explanation. We find that following such change patterns, during software maintenance, programmers can modify concurrent code to repair bugs, improve performance, and change functions of their code. Furthermore, we find that maintaining concurrent code is not a one-direction migration. Due to various considerations, programmers can apply seemingly contradictory changes, and even revert their changes. Sometimes, programmers can even make changes, before they fully understand the consequences of their changes.
- An application of our change patterns in real code. In particular, we search the latest versions of three projects for chances to apply our change patterns, and synthesize three pull requests according to our change patterns. Two of our pull requests are already confirmed and accepted by their programmers. However, our results also reveal that it needs much experience and understanding to leverage our change patterns.
- More findings on concurrent programs. For example, we reveal the trends of different parallel API classes. As another example, we find that at the granularity of commits, there still exists a correlation between total commits and concurrency-related commits, but it is less significant than at the granularity of code lines.

II. Methodology

This section presents the research questions (Section II-A), the data set (Section II-B), and the support tools (Section II-C) of our study.

A. Research questions

To understand how concurrent code is maintained, in this study, we focus on the following research questions:

RQ1. What change patterns are followed when programmers maintain concurrent code?

Based on their analysis on commits, Kim and Notkin [11] found that code changes can be repetitive, and Martinez *et al.* [16] further extracted change patterns to denote such repetitive changes. However, to the best of our knowledge, no previous study explored the change patterns of concurrent programming, and this research question is still largely open. In our empirical study, we summarize concurrency-related commits into five change patterns, and present examples to explain our change patterns (see Section III-A for details).

RQ2. How useful are our extracted change patterns, when programmers maintain concurrent code?

To assess the usefulness of our extracted change patterns, we search matching code in open-source projects with our change patterns. We manually exterminate return code. If a change pattern applies to such code, we fork the source files. After that we make our changes, and submit our pull request. Two of our pull requests are already confirmed and accepted by their programmers (see Section III-B for more details).

RQ3. What are the change trends of using parallel APIs?

Okur and Dig [20] found that in C#, 10% of parallel API classes account for 90% of API usages. In Java, J2SE[3] also provides standard APIs for developing concurrent programs. In this research question, we explore whether Java parallel APIs follow a similar distribution as C#. Our results show that in Java, popular parallel APIs also account for a large portion of the total API usages, but the distribution is less screwed (see Section III-C for details).

RQ4. Are there any correlations between total commits and concurrency-related commits?

At the granularity of code lines, Gu *et al.* [10] found a strong correlation between total changes and concurrency-related changes. With a different data set and a coarser granularity, we explore whether their finding still holds on our data set. Our results show that the correlation still exists, but is less significant (see Section III-D for details).

B. Dataset

In this study, we collected commits from six Apache[4] projects. Table I shows the details of our data set. We selected these projects, since they are all popular and active. These projects cover various types of projects such as distributed computing, web server, database, information retrieval, and network. In particular, Hadoop is one of the most popular distributed computing frameworks in Java. Tomcat is a popular web server. Cassandra is a database system that manages massive data. Solr is an enterprise search platform. Netty is an asynchronous network application framework. Flink is a stream processing framework. Column "#Commits" lists number of commits. Column "#Concurrency" lists number of concurrency-related commits. From these concurrency-related commits, we selected a subset for manual analysis. Column

[3]https://docs.oracle.com/javase/

[4]http://www.apache.org/

TABLE I
SELECTED COMMITS

Project	#Commits	#Concurrency	#Manual
Hadoop	14,930	2,739	64
Tomcat	17,731	1,963	207
Cassandra	21,982	1,904	78
Lucene-solr	26,152	2,375	99
Netty	7,759	1,387	210
Flink	9,771	1,500	38
Total	98,325	11,868	696

"#Manual" lists number of our selected commits. We checked out all the commits in December 2016. We next explain how to identify concurrency-related commits.

C. Study mechanism

As introduced in Section I, it is quite difficult to implement a single tool to automate our analysis. Instead, we employ and implement a set of tools to reduce the analysis effort. Inevitably, we have to introduce manual analysis in RQ1. Our study mechanism has the following steps:

1) Step 1. Collecting commits: All the projects in our study use Git[5] as their version control system. We implement a tool to check out all their commits. A typical commit log contains a commit id, an author name, the commit date, and a message. Once we get a commit id, our tool uses the `git show` command to list details, and then uses the textual `diff` command to produce its change hunks.

2) Step 2. Identifying commits for the follow-up analysis: From collected commits, the second step is to extract commits that are related to concurrent code. Here, we consider that a commit is related to concurrent programming, if the commit involves synchronization, thread, or concurrent API classes. In this paper, we call such commits as *concurrency-related commits*. A commit has a commit log that is written in natural language. The commit log often explains which files are modified and why programmers make such modifications. Our tool builds queries to search for commits that are related to concurrent programming. The built queries contain concurrency-related keywords. We choose 96 keywords as concurrency-related keywords. For example, `synchronized`, `volatile`, and concurrent API class names. The full list can be found in our project homepage[6]. However, this selector selects 12,427 commits that are too many for manual analysis. We selected a subset from them for manual analysis by checking whether a commit log contains concurrency-related keywords. The first selector selects 12,427 commits from ? commits. The second selector selects 561 commits from the 12,427 commits that are output of the first selector. The size of the final selected set is 561.

The textual matching method can lose some useful commits. We used a machine learning method to select concurrency-related commits from the 12,427 commits as supplements. Researchers have explored related problems. For example, Tian *et al.* [26] proposed an approach that identifies bug fixing patches with classification techniques. Motivated by their approach, we train a classifier to predict whether a commit is related to concurrent code. When training the classifier, our tool analyzes change hunks that are produced by the `diff` command, and uses the results as our code features. As shown in Table II, in total, our tool extracts 12 features from each commit. The first column shows the feature names, and the second column shows the explanations. The keywords are the same as the concurrency-related keywords used in the previous paragragh. Our tool employs the SVM [5] algorithm to identify commits that are related to concurrent programming. In particular, our tool is implemented based on a popular SVM library, LIBSVM [4]. As SVM is a supervised classification algorithm, it needs both labeled positive and negative data for training. We randomly selected 48 commits as a training set. We built the features of them, labelled them. The 48 commits have 15 positive instances and 35 negative instances. Then we trained a model and used it to classify commits. It selected 135 positive commits. The precision is ? based on a manual inspection. The accurate recall is unavailable.

TABLE II
FEATURES OF DATA

Feature	Explanation
msgKey	Number of keywords in commit message
file	Number of files in a commit
hunk	Number of hunks in a commit
lineAdd	Number of added lines in a commit
lineRemove	Number of removed lines in a commit
lineSub	lineAdd - lineRemove
lineSum	lineAdd + lineRemove
keyAdd	Number of added keywords in a commit
keyRemove	Number of removed keywords in a commit
keySub	keyAdd - keyRemove
keySum	keyAdd + keyRemove
contextKey	Number of keywords in context code

We selected 696 commits for manual analysis in total.

3) Step 3. Analyzing commits according to different research questions: We then conduct detailed analysis according to our research questions.

RQ1. Determining change patterns. To explore this research question, we analyzed each selected commit for their change patterns. For example, below is a concurrency-related commit. The top five lines describe the metadata of the commit. The other lines describe the differences between two versions of code.

```
1  commit 563e546236217dace58a8031d56d08a27e08160b
2  Author: zentol <s.motsu@web.de>
3  Date:   Mon Jan 26 11:07:53 2015 +0100
4  [FLINK-1419] [runtime] Fix: distributed ...
5
6  public FutureTask<Path> createTmpFile(...) {
7  -    synchronized (count) {
8         ...
9  +    synchronized (lock) {
10        ...
```

For each commit, we first read the metadata and the corresponding issue to understand why programmers make the commit. After that, we scan change hunks to understand the details. In a change hunk, the "+" symbol denotes added lines, and the "-" symbol denotes removed lines. In some cases, it is infeasible to determine the category of a commit based on only its change hunks. For example, as change hunks are limited,

[5]https://git-scm.com/

[6]https://github.com/qwordy/Research

it can be infeasible to determine the type of a variable. In such cases, we check out the original and modified versions of all files to analyze. In this example, we cannot determine the type of the `count` and `lock` variables. After we check out all the files, we understand that the types of them are `Map` and `Object`, respectively.

We classify concurrency-related commits into different categories, mainly according to our observed code changes such as the modifications on code elements, parallel libraries, and control flows.

RQ2. Exploring the usefulness of our change patterns. We prepare a set of keywords for each change pattern, and search Github[7] for code where the pattern can apply. For example, we use `synchronized`, `put` or `get` as keywords to search code pieces that manually handle synchronization of collections. We find numerous code pieces in the search results. We manually check the code and decide whether it is acceptable for the pattern. If it is, we fork the project; make our changes; and submit the pull request.

RQ3. Determining the trends of using parallel APIs. To explore this research question, we analyzed all the 53 classes that are declared in the `java.util.concurrent` package. We selected this package, since it is a popular package for concurrent programming. For each parallel API class, we count their occurrences in all the 11,868 concurrency-related commits. Here, we consider that a class occurs a concurrency-related commit, if the class occurs in added lines or deleted lines of the commit. In addition, we draw its trend of its occurrences in the interval of months. Based on occurrences, we put these classes into three categories, and based on their trends, we further put them into three subcategories.

RQ4. Determining the correlations between total commits and concurrency-related commits Gu *et al.* [10] find that at the granularity of code lines, concurrency-related modifications and total modifications have strong correlation. In our study, we further explore the problem at the granularity of commits, for all the 11,868 concurrency-related commits. In particular, we compare total commits and concurrency-related commits in the interval of months. Gu *et al.* [10] use the Spearman's rank correlation coefficient to determine the correlation. In our study, we use the identical measure. Given two sets of data, the Spearman's rank correlation coefficient calculates a correlation value r between -1 and +1. Cohen [**?**] defines, $0.1 < |r| < 0.3$ as small correlation, $0.3 \leq |r| < 0.5$ as moderate correlation, and $|r| \geq 0.5$ as strong correlation. Here, -1 also denotes a strong monotonic negative correlation.

III. Results

A. RQ1. Change patterns

Table III and Table IV show an overview of our extracted change patterns. The first column is the sequence number of patterns. The "source" column shows the concrete examples of patterns. We put related source code in it. The left is the original code and the right is the modified code. We align the corresponding statements. We use different colors to mark modified lines. The "pattern" column shows the extracted patterns. They are short as they ignore the specific statements.

1. Changing lock types. It is feasible to lock resources with different mechanisms. For example, Java has a keyword, `synchronized`. The keyword can lock a block of code lines. With the keyword, programmers do not have to acquire and release resources explicitly. Alternatively, programmers can explicitly lock resources with APIs (*e.g.*, `ReentrantLock`). Explicit locks offer more features than the `synchronized` keyword does. For example, with such APIs, programmers can determine the conditions for a lock. As another example, besides exclusive locks, programmers can use shared locks. When a thread is holding an exclusive lock, other threads have to wait until it is released. In contrast, shared locks allow multiple threads to hold the lock with specific actions. For example, `ReadWriteLock` allows multiple read actions, but denies multiple write actions.

We find that programmers can replace the `synchronized` keyword with parallel API classes. For example, the first item of Table III comes from YARN-5825[8]. To improve the performance, programmers replaced the `synchronized` with the `getReadLock` method. The method returns a shared lock, so multiple threads can read the query simultaneously.

Meanwhile, we find that programmers can replace parallel API classes with the `synchronized` keyword. For example, the second item of Table III comes from Tomcat[9]. This commit is not reported, and we find it through our SVM classifier.

```
A ReadWriteLock cannot be used to guard a WeakHashMap.
    The WeakHashMap may modify itself on get(), as it
    processes the reference queue of items removed by
    GC. Either a plain old lock / synchronization is
   needed, or some other solution.
```

As the above message explains, a developer complained that the `ReadWriteLock` method does not guard the `WeakHashMap` variable, since the `get()` method can modify the `WeakHashMap` variable, and the modification can bypass the lock. In this example, programmers fixed the problem by replacing the methods with the `synchronized` keyword.

2. Changing locked variables. A program needs to lock variables before it enters critical section bodies. During software maintenance, programmers can change locked variables, and we find that the main purpose is to repair bugs. For example, the third item of Table III comes from FLINK-1419. This bug complains that `DistributedCache` does not preserve files for subsequent operations. Based on its discussions, we understand that in the buggy file, programmers lock `count`, while they shall lock `lock`. To fully fix the bug, programmers also modified the critical sections and the `finally` clause.

As another example, when repairing bugs, programmers can add new locks. For example, the fourth item of Table III comes from Tomcat[10]. It includes the following message:

[7]https://github.com

[8]https://issues.apache.org/jira/browse/YARN-5825 To save space, we remove the URLs of the other Apache issues. Their URLs can be built by replacing the above URL with their issue number.

[9]https://svn.apache.org/viewvc?view=revision&revision=1414150

[10]https://bz.apache.org/bugzilla/show_bug.cgi?id=58386

TABLE III
CHANGE PATTERNS

#	Source Code		Simplified Code	
	Original	Modified	Original	Modified

1	LeafQueue leafQueue = ...;
	-synchronized (leafQueue) {
	
	  57 LOC
	
	}
		LeafQueue leafQueue = ...;
		+try {
		+ leafQueue.getReadLock().lock();
		  57 LOC
		+} finally {
		+ leafQueue.getReadLock().unlock();
		}
			synchronized (obj)
			{
			  ...
			}
				try {
				  obj.lock();
				  ...
				} finally {
				  obj.unlock();
				}

2	-Lock readlock =
	- classLoaderContainerMapLock.readLock();
	-try {
	- readlock.lock();
	- result = classLoaderContainerMap.get(tccl);
	-} finally {
	- readlock.unlock();
	-}
	-if (result == null) {
	- Lock writelock =
	- classLoaderContainerMapLock.writeLock();
	- try {
	- writeLock.lock();
	    result = classLoaderContainerMap.get(tccl);
	    if (result == null) {
	      result = new ServerContainerImpl();
	      classLoaderContainerMap.put(tccl,result);
	    }
	- } finally {
	- writeLock.unlock();
	- }
	}
		+synchronized (classLoaderContainerMapLock) {
		
		  result = classLoaderContainerMap.get(tccl);
		  if (result == null) {
		    result = new ServerContainerImpl();
		    classLoaderContainerMap.put(tccl,result);
		  }
		
		}
			try {
			  readLock.lock();
			  read operations
			} finally {
			  readLock.unlock();
			}
			try {
			  writeLock.lock();
			  write operations
			} finally {
			
			writeLock.unlock();
			}
				synchronized {
				  all operations
				}

3	private static final Object lock = new
	  Object();
	private Map<...> count = new HashMap<>();
	-synchronized (count) {
	- Pair<Job, String> key =
	- new ImmutablePair<>(jobID, name);
	
	- if (count.containsKey(key)) {
	- count.put(key, count.get(key) + 1);
	  } else {
	- count.put(key, 1);
	  }
	}
		private static final Object lock = new
		  Object();
		private Map<...> count = new HashMap<>();
		+synchronized(lock)
		+ if (!jobCounts.containsKey(jobID)) {
		+ jobCounts.put(jobID, new HashMap<>());
		+ }
		+ Map<String, Integer> count =
		+ jobCounts.get(jobID);
		+ if (count.containsKey(name)) {
		+ count.put(name, count.get(name) + 1);
		  } else {
		+ count.put(name, 1);
		  }
		}
			synchronized (obj1)
			{
			  ...
			}
				synchronized
				(obj2) {
				  ...
				}

4	-public boolean isAccessed() {
	  return this.accessed;
	}
		+public synchronized boolean isAccessed() {
		  return this.accessed;
		}
			void foo() {
			  ...
			}
				synchronized void
				foo() {
				  ...
				}

5	-synchronized (this.channelLookup) {
	- try{
	    lookupResponse = AkkaUtils.
	<JobManagerMessages.ConnectionInformation>ask(ch
	annelLookup, new
	JobManagerMessages.LookupConnectionInformation(c
	onnectionInfo, jobID, sourceChannelID),
	timeout).response();
	- }catch(IOException ioe) {
	- throw ioe;
	- }
	-}
		lookupResponse = AkkaUtils.
		<JobManagerMessages.ConnectionInformation>ask(
		channelLookup, new
		JobManagerMessages.LookupConnectionInformation
		(connectionInfo, jobID, sourceChannelID),
		timeout).response();
			synchronized (obj)
			{
			  ...
			}
				...

6	synchronized (buffers) {
	  if (...) {
	- if (spillWriter != null) {
	- spillWriter.close();
	- }
	    isFinished = true;
	  }
	}
		synchronized (buffers) {
		  if (...) {
		    isFinished = true;
		  }
		}
		+if (spillWriter != null) {
		+ spillWriter.close();
		+}
			synchronized (obj)
			{
			  statements1
			  statements2
			}
				synchronized (obj)
				{
				  statements2
				}
				statements1

7	-public synchronized void reset() {
	
	  map.clear();
	  members = EMPTY_MEMBERS;
	
	}
		+private final Object membersLock = new
		+ Object();
		+public void reset() {
		+ synchronized (membersLock) {
		    map.clear();
		    members = EMPTY_MEMBERS;
		+ }
		}
			synchronized void
			foo() {
			  ...
			}
				void foo() {
				  synchronized
				(obj) {
				    ...
				  }
				}

TABLE IV
CHANGE PATTERNS (CONT.)

#	Source Code		Simplified Code	
	Original	Modified	Original	Modified

8	-synchronized void enqueue(final long seqno,
	final boolean lastPacketInBlock,
	final long offsetInBlock) {
	- if (running) {
	final Packet p = new Packet(...);
	LOG.debug(...);
	
	ackQueue.addLast(p);
	notifyAll();
	}
	}
		+void enqueue(final long seqno,
		final boolean lastPacketInBlock,
		final long offsetInBlock) {
		
		final Packet p = new Packet(...);
		LOG.debug(...);
		+ synchronized (this)
		+ if (running) {
		ackQueue.addLast(p);
		notifyAll();
		}
		}
			synchronized void
			foo(...) {
			statements1
			statements2
			}
				statements1
				synchronized (obj)
				{
				statements2
				}

9	-Membership membership = null;
	public boolean hasMembers() {
	if (membership == null) setupMembership();
	return membership.hasMembers();
	}
	synchronized void setupMembership() {
	if (membership == null) {
	membership = new Membership(
	super.getLocalMember(true));
	}
	}
		+volatile Membership membership = null;
		public boolean hasMembers() {
		if (membership == null) setupMembership();
		return membership.hasMembers();
		}
		synchronized void setupMembership() {
		if (membership == null) {
		membership = new Membership(
		super.getLocalMember(true));
		}
		}
			T foo;	volatile T foo;

10	-private volatile int requestCount; -private volatile int errorCount; - requestCount++;	+private final AtomicInteger requestCount = + new AtomicInteger(0); +private final AtomicInteger errorCount = + new AtomicInteger(0); + requestCount.incrementAndGet();	volatile T foo;	TT foo;

```
1  Reported by RV-Predict (a dynamic race detector) when
       running the test suite:
2  Data race on field org.apache.catalina.tribes.io.
       ObjectReader.accessed: {{{
3  Concurrent write in thread T93 (locks held: {...})
```

The data race indicates that the `isAccessed()` is not locked, so programmers add the `synchronized` keyword to allow locking on the method.

We find that programmers can also delete unnecessary locks. For example, the fifth item of Table III comes from Flink:

```
1  ...Remove synchronized blcok in getReceiverList of
       ChannelManager which effectively serialized the
       connection lookup calls of a single task manager.
```

As the above commit messages says, programmers removed synchronization from the `ChannelManager` class to improve the performance.

In some other cases, we find that programmers can refine their locked resources to improve performance. For example, the seventh item of Table III comes from Tomcat[11]. The original code locks the instance of a class, but the modified code locks only the `membersLock` field.

3. Modifications inside critical section bodies. A critical section is a code block that is executed, when a thread locks the corresponding resources. We notice that even modifications inside critical section bodies can repair concurrency bugs. For example, the sixth item of Table IV comes from FLINK-2384. It is caused by an implicit lock in the `spillWriter.close()` method. As programmers typically do not known such locks inside APIs, the lock leads to the deadlock. Indeed, we find that a recent benchmark [13] includes a similar concurrency bug, and Lin *et al.* [14] proposed an approach that detects such implicit locks inside APIs. In this example, programmers move the `spillWriter.close()` method outside the critical section body to resolve the deadlock.

Besides the above example, the majority of modifications on critical section bodies indicates new functionalities or refactoring. For example, the eighth item of Table IV comes from HDFS-4200. To reduce the size of a critical section body, programmers refactor the body into several methods. Indeed, this issue involves modifications that are related to even more change patterns (*e.g.*, adding locked variables).

4. Changing the `volatile` keyword. In Java, the `volatile` keyword denotes a variable that is saved in the main memory. For a `volatile` variable, a thread reads its latest value from the main memory, instead of the obsolete value in the cache. Although it can improve the overall performance, races can occur, when multiple threads read and write `volatile` variables simultaneously.

We find that programmers can add the `volatile` keyword to improve performance. For example, the ninth item of Table IV comes from Tomcat[12]. It reports a data race, and the bug was repaired by adding the `volatile` keyword.

Besides adding the keyword, we find that programmers have to remove the `volatile` keyword, since they do not fully understand its meanings. For example, in tenth item of Table IV comes from Tomcat[13]. In this bug, programmers auto-increment a `volatile` variable, but such an action is not atomic. As a result, FindBugs reports it as a bug, and programmers have to remove the keyword.

[11]https://bz.apache.org/bugzilla/show_bug.cgi?id=58382

[12]https://bz.apache.org/bugzilla/show_bug.cgi?id=58392

[13]https://svn.apache.org/viewvc?view=revision&revision=1360946

5. Replacing self-written code with Parallel APIs. Programmers can implement concurrent code by themselves, but later they realize that it is easier to call APIs that already implement their functionalities. For example, a previous version of Hadoop has the following code:

```
private volatile long genstamp;
public synchronized long nextStamp() {
  this.genstamp++;
  return this.genstamp; }
```

Later, a programmer realized that it is better to replace the above code with the `AtomicLong` class. He reported the issue (HDFS-4029), and set its priority as major. Here, `AtomicLong` is a thread-safe version of type long. It allows updating a `Long` value without explicit synchronization and it is fast.

```
private volatile long genstamp;
public long nextStamp() {
  return genstamp.incrementAndGet(); }
```

Besides replacing directly, programmers can replace their code with parallel APIs that implement similar functions. For example, the below code comes from LUCENE-2779:

```
  protected HashMap<String,RAMFile> fileMap = ...;
  public final boolean fileExists(String name) {
    ensureOpen();
    RAMFile file;
    synchronized (this) {
      file = fileMap.get(name);
    }
    return file != null;
}
```

Instead of handling synchronization by themselves, programmers replaced the above code with a parallel API:

```
 protected Map<String,RAMFile> fileMap = ...;
 public final boolean fileExists(String name) {
    ensureOpen();
    return fileMap.containsKey(name);
 }
```

In summary, in our study, we identify five types of change patterns in total. First, we find that programmers can modify parallel keywords and locked variables, and such modifications are mainly for repairing bugs and improving performance. Second, we notice modifications on critical section bodies, and such modifications often indicate new functions. Finally, we notice that self-written code is replaced with corresponding parallel APIs. In most cases, we find that maintaining concurrent code is not a one-direction migration. For example, there are several different ways to implement a lock. Programmers shall carefully analyze their programming contexts to determine which is their best choice. As shown in our examples, during software maintenance, programmers can migrate to any types according to their need.

B. RQ2. The usefulness of our change patterns.

In total, we made three pull requests. We made the first pull request on Schmince-2[14]. This is a game, where players control space tourists for adventure. It has the following code:

```
public class DRandom {
  private ... random = new ThreadLocal<Random>() {
    protected Random initialValue() {
      return new Random();
    }
  };
  public static Random get() {
    return random.get();
  }
}
```

The above code implements a class that generates random values for multiple threads. We find that J2SE provides the `ThreadLocalRandom` class[15] that implements the identical function. According to our fifth change pattern, we made a pull request to replace the above code with the corresponding API[16], and the modified code is as follow:

```
public class DRandom {
  public static Random get() {
    ThreadLocalRandom.current();
  }
}
```

This pull request is already confirmed by programmers of the project.

We made the second pull request on UnifiedEmail[17]. It is an Android email client. It has the following code:

```
private static NotificationMap sActiveNotificationMap
     = null;
private static synchronized NotificationMap
     getNotificationMap(Context context) {
 if (sActiveNotificationMap == null) {
  sActiveNotificationMap = new NotificationMap();
  sActiveNotificationMap.loadNotificationMap(context);
 }
 return sActiveNotificationMap;
}
```

If the `sActiveNotificationMap` field is `null`, the above method creates a new map and assigns the new map to the field. To allow multiple threads to call the methods, programmers add the `synchronized` keyword to the method. We believe that when `sActiveNotificationMap` is not `null`, the lock is unnecessary, since it does not change the field. According to our second change pattern, we made the following modification to synchronize only the lines that modify the field:

```
private static volatile NotificationMap
     sActiveNotificationMap = null;
private static NotificationMap getNotificationMap(
     Context context) {
  if (sActiveNotificationMap == null) {
    synchronized (NotificationUtils.class) {
      if (sActiveNotificationMap == null) {
       sActiveNotificationMap = new NotificationMap();
       sActiveNotificationMap.loadNotificationMap(
            context);
      }
    }
  }
  return sActiveNotificationMap;
}
```

The owner of the project deleted our pull request. We checked the pull requests of the project and found that there is no open or closed pull request. This project has more than 19,000 commits. It is possible that it has a strict policy of introducing external source code.

[14]https://github.com/derekmu/Schmince-2

[15]https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadLocalRandom.html

[16]https://github.com/derekmu/Schmince-2/pull/1

[17]https://github.com/HexagonRom/android_packages_apps_UnifiedEmail

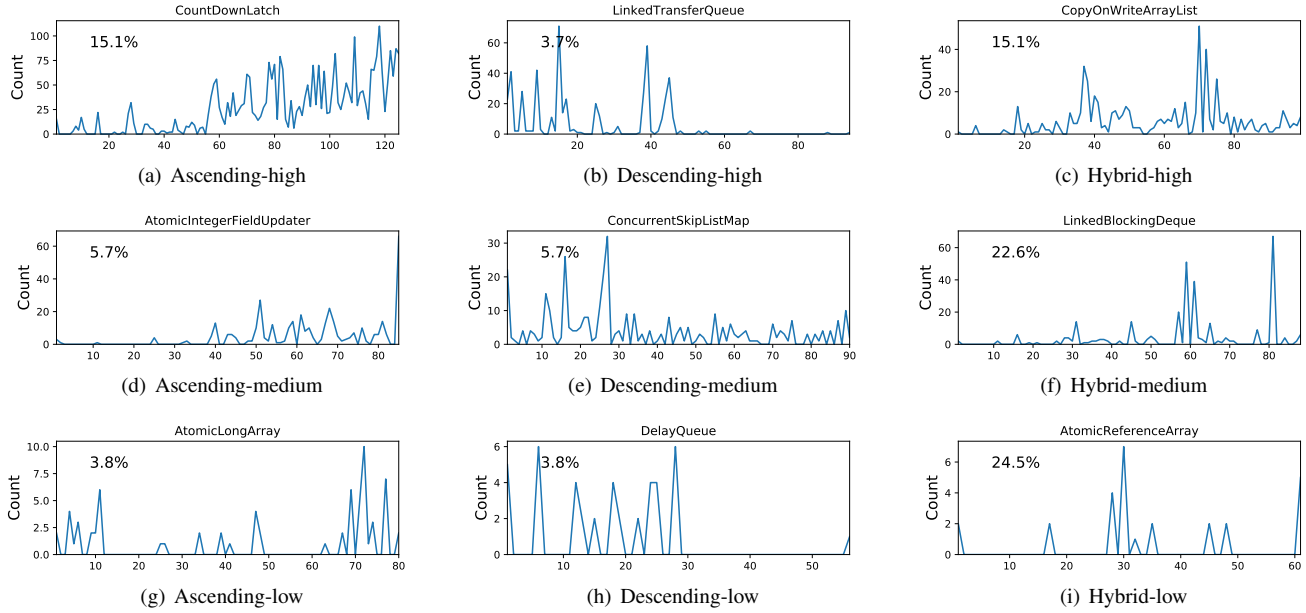


Fig. 1. Typical trends and their percents

TABLE V
TOP 10 ACTIVE CLASSES

Class	Occurrence
AtomicInteger	6,891
AtomicBoolean	4,180
ConcurrentHashMap	3,976
AtomicLong	3,926
CountDownLatch	3,211
AtomicReference	2,254
Executors	2,026
ThreadPoolExecutor	1,617
LinkedBlockingQueue	1,553
ConcurrentLinkedQueue	1,435

We made the third pull request on Spider4java[18]. It is a Java web crawler. It has the following code:

```
1  public class Counter {
2    protected int count;
3    public Counter() {
4      count = 0;
5    }
6    public synchronized void increment() {
7      count = count + 1;
8    }
9    public synchronized int getValue() {
10     return count;
11   }
12 }
```

The above class implements a counter for multiple threads. As J2SE provides the identical API, `AtomicInteger`[19], we modify it as follow:

```
1  public class Counter {
2    protected AtomicInteger count;
3    public Counter() {
4      count = new AtomicInteger();
5    }
6    public void increment() {
7      count.getAndIncrement();
8    }
9    public int getValue() {
10     return count.get();
11   }
12 }
```

This pull request has been accepted.

In summary, our results show that our change patterns are repetitive in future maintenance, and programmers confirmed that our change patterns are useful. However, our results also reveal that it needs much programming experience to fully unleash the potential of our change patterns. We further discuss this issue in Section IV.

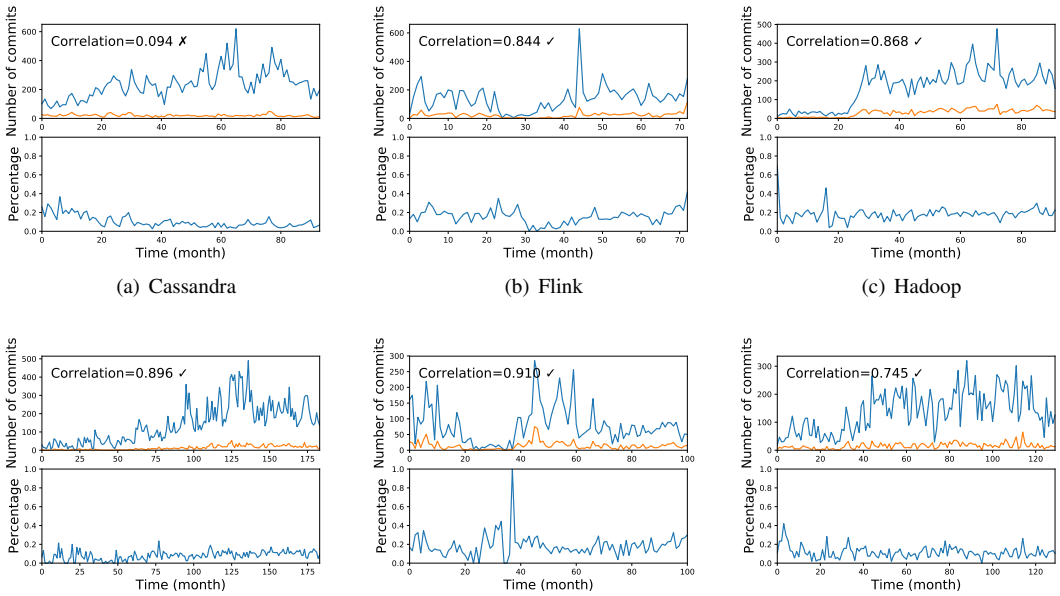
C. RQ3. What are the change trends of using parallel APIs

Table V shows the top ten active parallel API classes in Java. Column "class" lists names of these classes. Column "Occurrence" lists their occurrences in our concurrency-related commits. Dig [20] found that in C#, 10% of parallel API classes account for 90% of API usages. We find that Java follows similar usages patterns as C#. However, the distribution of Java is not as screwed as C#. We find that the top %10 classes account only for about 50% of the total occurrences, while the bottom %10 classes account for 0.36%.

Based on their occurrences, we put the 53 classes into three categories such as *high-frequency*, *medium-frequency* and *low-frequency*, and each category contains roughly the same number of classes. For each category, we further put its classes into three subcategories such as *ascending*, *descending* and *hybrid*. Figure 1 shows the results. The percent in each chart is calculated from the classes with the corresponding trend to the total classes. Although most classes are in the hybrid trend, we notice that in the high-frequent category, 15.1% of classes are in the ascending subcategory. The trend can indicate that about 1/4 parallel API classes in Java standard libraries are becoming more popular.

[18]https://github.com/lichangshu/spider4java

[19]https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html



(a) Cassandra (b) Flink (c) Hadoop

(d) Lucene-Solr (e) Netty (f) Tomcat

Fig. 2. Number of concurrency-related commits compared to all commits

In summary, we find that Java parallel API classes follow a similar distribution as C#, but the distribution is less screwed. However, the trends indicate that the distribution can become more like C#, since popular ones are becoming more popular.

D. RQ4. The correlations between total commits and concurrency-related commits.

Figure 2 shows our results. Each label denotes a project, and has two associated figures. In particular, the upper figure denotes total commits and concurrency-related commits in the interval of months. Its horizontal axis denotes time, and its vertical axis denotes number of commits. Its blue curve denotes total commits, and its yellow curve denotes concurrency-related commits. We find that the curves of total commits and concurrency-related commits are not quite similar. However, the Spearman's rank correlation coefficient shows that in the five out of six projects, the two sets of values are correlated. One exception is Cassandra, where the correlation value is at the border (0.094).

In the lower figures, we calculate the percents from total commits to concurrency-related commits. Although the two sets of data are correlated, we find that their percents are inconstant. In total, about 20% of total commits are related to concurrency. However, if we make predication based on the percents, the prediction can be unreliable.

In summary, we find that at a coarser granularity, the correlation between total commits and concurrency-related commits still exists with a strong correlation in general.

E. Threats to Validity

The threats to internal validity include that our tool can omit some concurrency commits. Due to various issues, our tool can fail to identify all concurrency commits. To reduce the threat, we employ both the query-based search and a classifier in our study. The threat could be further reduced by more advanced identification techniques. The threats to internal validity also include obsolete commits. With the rapid development of software, such commits may present obsolete or even wrong usages. To reduce the threat, in our study, we prefer to recent commits. The threats to external validity include our selected projects and programming language. The number of the projects we select is small. They are all Java-based Apache projects. The threat could be reduced by introducing more projects and languages in future work.

IV. Discussion and Future Work

The extension of existing tools. Our patterns and findings can help improving existing program transformation tools. For example, Lin *et al.* [12] proposed Asynchronizer that transforms self-written code with the `AsyncTask` class, and Tao and Qian [25] support three our found patterns through refactoring. With minor modifications, we can extend the above tools to transform more APIs and to support more patterns. We plan to explore such extension in future work.

The usefulness of our other findings. In our empirical study, we reveal the usefulness of our change patterns, but we do not explore the benefits of our other findings. We believe that our other findings can also be useful in specific scenarios. For example, the trend of a specific class can have correlations with other factors such as the changes inside APIs. In future work, we plan to explore such factors, so that we can explore the usefulness of our other findings.

V. Related Work

Empirical studies on concurrent programming. In literature, researchers have conducted various empirical studies to

understand concurrent programming. Pinto *et al.* [22] conducted a large scale study on the usage of concurrency in Java, and Wu *et al.* [28] replicated their study with C++. Okur and Dig [20] studied how developers use parallel libraries in C#, and their results reveal some interesting findings. For example, at least 10% of programmers misuse libraries, so their code runs sequentially rather than concurrently. David *et al.* [7] conducted an empirical study to investigate synchronization at both hardware and software levels, and their results show that the scalability of synchronization is mainly a property of hardware. Pinto *et al.* [21] analyzed concurrency-related threads of StackOverflow, and they found that most threads discussed basic questions (*e.g.*, "what is a mutex?"). Sadowski *et al.* [23] studied the evolution of data races, and they found that many data races always exist. Xin *et al.* [30] conducted an empirical study on lock usage, and they found that most functions acquire only a lock. Lu *et al.* [15] studied characteristics of real world concurrency bugs. The above approaches do not analyze change patterns and the trends of parallel APIs, which are complemented by our study.

Identification of commits. Zhong and Su [31] relied on simple heuristic to identify bug fixes from commits. Tian *et al.* [26] trained a classifier to identify bug fixes based on their extracted features. Wu *et al.* [29] built the links between bug fixes and their reports based their similarity values. The above approaches focus on identify bug fixes from commits. In our study, we implement a tool that identifies concurrency-related commits, complementing the above approaches.

VI. Conclusion

Concurrent programming is challenging, and a mistake can introduce hidden bugs that are difficult to be detected. During software maintenance, programmers have to handle concurrent code carefully. Researchers have conducted various empirical studies to understand concurrent programming. However, how programmers maintain concurrent code is still rarely studied. In this paper, we conduct an empirical study to understand the change patterns and other perspectives of concurrent programming. Based on our analysis results, we summarize five change patterns. We show that such change patterns are repetitive in future maintenance, and programmers have confirmed the usefulness of our extracted patterns. Furthermore, our study comes to other findings such as the trends of parallel API classes and the correlation between total commits and concurrency-related commits, whose usefulness is worthy of further exploration in future work.

References

[1] Y. Ahn, J. Suh, S. Kim, and H. Kim. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):71–85, 2003.

[2] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *ESEC/FSE*, pages 385–394, 2007.

[3] H. Borges. On the popularity of github software. In *ICSME*, page 618, 2016.

[4] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[5] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[6] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *OOPSLA*, pages 313–328, 2008.

[7] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.

[8] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Transactions on Software Engineering*, 33(11):725–743, 2007.

[9] D. M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[10] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu. What change history tells us about thread synchronization. In *ESEC/FSE*, pages 426–438, 2015.

[11] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.

[12] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for android applications through refactoring. In *FSE*, pages 341–352, 2014.

[13] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao. JaConTeBe: A benchmark suite of real-world Java concurrency bugs. In *ASE*, pages 178–198, 2015.

[14] Z. Lin, H. Zhong, Y. Chen, and J. Zhao. LockPeeker: detecting latent locks in Java APIs. In *ASE*, pages 368–378, 2016.

[15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[16] M. Martinez, L. Duchien, and M. Monperrus. Automatically extracting instances of code change patterns with AST analysis. In *ICSME*, pages 388–391, 2013.

[17] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the android ecosystem. In *ICSM*, pages 70–79, 2013.

[18] P. E. McKenney. Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a). *CoRR*, abs/1701.00854, 2017.

[19] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[20] S. Okur and D. Dig. How do developers use parallel libraries? In *FSE*, page 54, 2012.

[21] G. Pinto, W. Torres, and F. Castor. A study on the most popular questions about concurrent programming. In *PLATEAU@SPLASH*, pages 39–46, 2015.

[22] G. Pinto, W. Torres, B. Fernandes, F. C. Filho, and R. S. M. de Barros. A large-scale study on the usage of java's concurrent programming constructs. *Journal of Systems and Software*, 106:59–81, 2015.

[23] C. Sadowski, J. Yi, and S. Kim. The evolution of data races. In *MSR*, pages 171–174, 2012.

[24] G. Santos, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente. System specific, source code transformations. In *ICSME*, pages 221–230, 2015.

[25] B. Tao and J. Qian. Refactoring Java concurrent programs based on synchronization requirement analysis. In *Proc. ICSME*, pages 361–370, 2014.

[26] Y. Tian, J. Lawall, and D. Lo. Identifying Linux bug fixing patches. In *ICSE*, pages 386–396, 2012.

[27] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 2016.

[28] D. Wu, L. Chen, Y. Zhou, and B. Xu. An extensive empirical study on C++ concurrency constructs. *Information & Software Technology*, 76:1–18, 2016.

[29] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *ESEC/FSE*, pages 15–25, 2011.

[30] R. Xin, Z. Qi, S. Huang, C. Xiang, Y. Zheng, Y. Wang, and H. Guan. An automation-assisted empirical study on lock usage for concurrent programs. In *ICSM*, pages 100–109, 2013.

[31] H. Zhong and Z. Su. An empirical study on real bug fixes. In *ICSE*, pages 913–923, 2015.