Received July 13, 2019, accepted July 26, 2019, date of publication July 30, 2019, date of current version August 14, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2931953

Automated Refactoring for Stampedlock

YANG ZHANG[1], SHICHENG DONG[1], XIANGYU ZHANG[2], HUAN LIU[1], AND DONGWEN ZHANG[1]
[1]School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050000, China
[2]Department of Computer Science, Purdue University, West Lafayette, IN 47906, USA

Corresponding author: Yang Zhang (zhangyang@hebust.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61440012, in part by the Scientific Research Foundation of Hebei Educational Department under Grant ZD2019093, and in part by the Fundamental Research Foundation of Hebei Province under Grant 18960106D.

ABSTRACT StampedLock, proposed in JDK 1.8, provides many interesting features, such as optimistic read locks and upgrading/downgrading locks to improve the design of concurrent programs instead of employing pure read/write locks. Existing refactorings have proposed algorithms to convert locks, but there are a few refactorings that use these promising features of StampedLock. To illustrate a possible refactoring, this paper first shows three code transformations based on StampedLock. Then, this paper presents CLOCK, an automated refactoring tool that helps developers convert the synchronized lock into the StampedLock. An algorithm for reentrance analysis is proposed for the precondition validation. The write lock, read lock, optimistic read lock, and upgrading/downgrading lock are inferred and refactored. CLOCK is evaluated with the SPECjbb2005 benchmark and two real-world applications, Xalan and FOP. A total of 66 classes are modified by searching approximately 395KSLOC and applying the refactoring, achieving an average of 22 classes per benchmark. The experimental results show that CLOCK can help developers with refactoring for StampedLock and save developer effort.

INDEX TERMS Automated refactoring, StampedLock, reentrance analysis, optimistic read lock, upgrading/downgrading lock.

I. INTRODUCTION

Lock, as one of the synchronization mechanisms, is used to ensure the correctness of shared resources access in concurrent programs. Commonly, a lock provides exclusive access to a shared resource so that only one thread at a time can acquire the lock while other threads will have to wait for the release of the lock. Locks are widely used, but they suffer from some problems, such as deadlock, livelock, priority reversion, convoying and lock contention. Among them, lock contention often leads to poor scalability and low performance, which comprise as a main challenge in the multi-core era.

Various synchronization mechanisms, such as software transactional memory (STM) [1], [2] and the lock-free algorithm (LFA) [3], also exist for concurrent programming. Similar to locks, they have benefits and drawbacks. STM and LFA provide non-blocking execution and seem very suited for concurrent programs running on a multi-core/many-core processor. However, STM is not applicable if I/O or other irreversible operations exist. Furthermore, for those concurrent programs with a frequent data race, most transactions will have to roll back and start over, likely resulting in poor performance. Analogous to STM, programs with the LFA run continuously without blocking, making full utilization of multi-core processors. However, designing a correct and high-performance LFA usually requires expertise and seems difficult for typical developers. Although locks are susceptible to lock contention and other promising techniques for synchronization mechanisms (e.g., STM and LFA) have been proposed, it seems that locks will continue to be used in the future.

Java has provided several locks, such as the synchronized lock, *ReentrantLock*, *ReentrantReadWriteLock*, and *StampedLock*. Early in JDK1.0, the synchronized lock was introduced as a synchronized method or a synchronized block. With an implicit monitor object and without explicit release operations, developers can use and understand this lock easily. Since JDK1.5, Java has introduced both *ReentrantLock* and *ReentrantReadWriteLock* in the

The associate editor coordinating the review of this manuscript and approving it for publication was Mohammad Anwar Hossain.

java.util.concurrent.locks package to support an explicit locking mechanism. *ReentrantLock* has the same behavior and semantics as the synchronized lock but provides extended capabilities, such as a fairness strategy, trying to acquire a lock and interrupting lock acquisition. *ReentrantReadWriteLock* provides read and write locks. It allows the read lock to be held simultaneously by multiple reader threads as long as there are no writers, and the write lock is exclusive. However, *ReentrantReadWriteLock* may suffer from severe starvation if there are a number of reads but very few writes. The fairness strategy may help improve this problem but may compromise throughput [4]. With JDK1.8, *StampedLock* was introduced and involved many interesting features that were unsupported by previous Java locks, such as optimistic read, upgrading/downgrading lock, and acquiring or releasing a lock with a *stamp* value. Similar to *ReentrantReadWriteLock*, *StampedLock* also has read and write locks by the methods *asReadLock()* and *asWriteLock()*, but it does not seem to suffer from severe starvation.

Both academic and industrial sectors have proposed techniques and tools for refactoring among locks in the past few years. In the academic community, Schäfer *et al.* [5] proposed a refactoring tool *Relocker* with the techniques of converting the synchronized lock to a *ReentrantLock* and of converting a *ReentrantLock* to a *ReentrantReadWriteLock*. With *Relocker*, developers can easily select a relatively high-performance lock by tuning the performance among these locks. Tao et al. [6] proposed an automated refactoring approach for Java concurrent programs based on synchronization requirement analysis. Their approach could find refactoring opportunities for splitting locks and converting locks to atomic operations. In the industrial setting, some commercial refactoring tools, such as concurrency-oriented refactoring for JDT [7] and LockSmith [8], were integrated into the modern integrated development environment (IDE) *IntelliJ IDEA* and *Eclipse* respectively. Both tools could split and merge locks, convert among locks, and make the field atomic.

Although many techniques and tools have been proposed to convert locks, we are not aware of any works that perform refactoring for *StampedLock*. The optimistic read lock and upgrading/downgrading lock provided by *StampedLock* provide an alternative for developers to improve the design of concurrent programs. Therefore, we attempt to work on the refactoring for *StampedLock* by taking advantage of these advanced locking operations.

Refactoring from the synchronized lock to *StampedLock* is non-trivial and challenging. First and foremost, the behavior semantic of *StampedLock* is different from the synchronized lock. The synchronized lock is reentrant while *StampedLock* is not. Hence, we need to check reentrance and avoid such refactoring if reentrance occurs. Second, the synchronized lock only uses the synchronized keyword while *StampedLock* owns multiple lock modes, such as the write lock, read lock and optimistic read lock. Determining how to infer these locks accurately requires more program analysis. Third, compared with inferring the read/write lock, inferring the upgrading/downgrading lock is more difficult. Automated refactoring tools need to know where and how to upgrade/downgrade a lock.

To meet these challenges, this paper focuses on refactoring support for the advanced locking operations, e.g., upgrading/downgrading locks and optimistic read locks provided by *StampedLock*. We present CLOCK, an automated refactoring tool that helps developers with refactoring the synchronized lock to *StampedLock*. We propose an algorithm for reentrance analysis as the precondition validation and the regulation for inferring the write lock, read lock, optimistic read lock and the upgrading/downgrading lock. CLOCK is evaluated with the SPECjbb2005 benchmark and two real world applications Xalan and FOP. A total of 66 classes are modified by searching among approximately 395KSLOC and applying the refactoring with an average of 22 classes per benchmark. Experimental results show that CLOCK can help the developer with refactoring and save developer effort.

This paper makes the following contributions:

- We describe the novel problem of converting the synchronized lock to *StampedLock*.
- We design a detection algorithm for reentrancy and inferring regulation for locks.
- A prototype tool, named CLOCK, is implemented as an extension to Eclipse IDE.
- CLOCK is evaluated on several Java benchmarks and applications, demonstrating the effectiveness of CLOCK.

The rest of this paper is organized as follows. Section II presents three motivating examples. Section III presents our refactoring framework and some details. Some practical problems are considered in Section IV, and a screenshot of CLOCK is shown in Section V. Section VI shows the experimental evaluation of the refactoring. The related works of literature are examined in Section VII and conclusions are drawn in Section VIII.

II. MOTIVATING EXAMPLES

This section presents three motivating examples to demonstrate the rationale. These examples show a variety of possible improvements of lock usage that may be introduced by *StampedLock*

Figure 1 shows three implementations of a method *delete()* based on the built-in monitor, ReadWriteLock and *StampedLock*. In this method, it first validates if the value *key* already exists or not, and then deletes the value *key* if it exists. This is a common practice for data structures to remove an element.

Figure 1(a) presents the method *delete()* based on the built-in monitor. The *synchronized* modifier is added to the method declaration to make it synchronized. If this method is converted to the read-write lock via *Relocker* [5], a write lock will be inferred as shown in Figure 1(b), because *Relocker* finds the side-effect of the method *remove()*. However, we notice that the method *remove()* will be executed only if the node exists. If the node does not exist, this method is not executed at all. Only the method *contain()* is executed and has

```
1 public synchronized void delete(int key){
2     boolean hasNode = contain(key);
3     if(hasNode){
4         remove(key);
5     }
6 }
```

(a) Method *delete()* based on built-in monitor

```
1  public void delete(int key){
2      ReadWriteLock lock = new ReentrantReadWriteLock();
3      lock.writeLock.lock();
4      try{
5          boolean hasNode = contain(key);
6          if(hasNode){
7              remove(key);
8          }
9      }finally{
10         lock.writeLock.unlock();
11     }
12 }
```

(b) Method *delete()* based on ReadWriteLock

```
1   public void delete(int key){
2     StampedLock slock = new StampedLock();
3     long stamp = slock.readLock();
4     try{
5       boolean hasNode = contain(key);
6       if (hasNode) {
7         long ws = slock.tryConvertToWriteLock(stamp);
8         if(ws! = 0L){
9           stamp = ws;
10        }else{
11          slock.unlockRead(stamp);
12          stamp = slock.writeLock();
13        }
14        remove(key);
15      }
16    } finally {
17        slock.unlock(stamp);
18    }
19  }
```

(c) Method *delete()* based on StampedLock

FIGURE 1. Method *delete()* based on built-in monitor, ReadWriteLock and *StampedLock*. (c) shows how to upgrade a read lock to a write lock.

```
1   int length;
2   public synchronized void computeAndGet(){
3       length = ...;
4       int a = length;
5       int b = length;
6       int c = length;
7       ... // other read operations
8   }
```

(a) Method *computeAndGet()* based on built-in monitor

```
1  int length;
2  public void delete(int key){
3      ReadWriteLock lock = new ReentrantReadWriteLock();
4      lock.writeLock.lock();
5      try {
6          length = ...;
7          int a = length;
8          int b = length;
9          int c = length;
10          ... // other read operations
11     } finally{
12         lock.writeLock.unlock();
13     }
14 }
```

(b) Method *computeAndGet()* based on ReadWriteLock

```
1   int length;
2   public void computeAndGet() {
3     StampedLock slock = new StampedLock();
4     long stamp = slock.writeLock();
5     length = ... ;
6     try{
7       long ws= slock.tryConvertToReadLock(stamp);
8       if(ws != 0 ){
9         stamp = ws;
10      }else{
11        slock.unlockWrite();
12        stamp = slock.readLock();
13      }
14      int a = length;
15      int b = length;
16      int c = length;
17      ... // other read operations
18    } finally{
19      slock.unlock();
20    }
21  }
```

(c) Method *computeAndGet()* based on StampedLock

FIGURE 2. Method *computeAndGet()* based on built-in monitor, ReadWriteLock and *StampedLock*. (c) shows how to downgrade a write lock to a read lock.

no side-effect. Therefore, a read lock should be inferred since it may introduce more concurrency. This situation often occurs for those data structures in which most of the nodes do not exist, especially for a newly created data structure. Hence, for this code segment, we should first use a read lock to test the existence of the value *key*, and only when the conditional statement becomes true will the write lock be used.

Figure 1(c) shows an alternative by taking advantage of the upgrading lock of *StampedLock*. It first employs a read lock (Line 3), and then judges if the value *key* exists or not. It upgrades this lock to a write lock if the value *key* exists (Lines 7-13). It leverages the method *tryConvertToWriteLock()* of the class *StampedLock* to perform the state conversion of the lock *slock*. If the conversion succeeds, a valid stamp *ws* will be returned and the argument *stamp* will be updated (Line 9). Even if the method *tryConvertToWriteLock()* fails, it may release the read lock (Line 11) and acquire a write lock (Line 12) to remove the value *key*. When releasing the lock, it uses the method *unlock()* of the class *StampedLock* with the value *stamp*.

Figure 2 shows three implementations of the method *computeAndGet()* based on the built-in monitor, ReadWriteLock and *StampedLock*. This method first computes the value of the field *length*, and then assigns the value *length* to the local variant *a*, *b* and *c*.

Figure 2(a) shows the method *computeAndGet()* based on the built-in monitor. When this method is refactored to *ReadWriteLock* by *Relocker* [5], as shown in Figure 2(b), a write

```
1   int length;
2   public synchronized int getLength(){
3         int temp = length;
4         return temp;
5   }
```
(a) Method *getLength()* based on built-in monitor

```
1  int length;
2  public void delete(int key){
3      ReadWriteLock lock = new ReentrantReadWriteLock();
4      lock.readLock.lock();
5      try {
6          int temp = length;
7          return temp;
8      } finally {
9          lock.readLock.unlock();
10     }
11 }
```
(b) Method *getLength()* based on ReadWriteLock

```
1   int length;
2   public int getLength(){
3     StampedLock slock = new StampedLock();
4     long stamp = slock.tryOptimisticRead();
5     int temp = length;
6     if (!slock.validate(stamp)) {
7        stamp = slock.readLock();
8        try {
9           temp = length;
10          return temp;
11       } finally {
12          slock.unlockRead(stamp);
13       }
14    }
15  }
```
(c) Method *getLength()* based on StampedLock

FIGURE 3. Method *getLength()* based on built-in monitor, ReadWriteLock and *StampedLock*. (c) shows how to convert a built-in monitor to an optimistic read lock.

lock will be inferred since *Relocker* finds the side-effect that the field *length* will be updated. However, the method *computeAndGet()* only has an update statement (line 3) followed by several field-read statements (lines 4-6). It may have a performance penalty for those read operations when it still holds a write lock. Developers can attempt to downgrade a write lock to a read lock after the update operation is executed.

Figure 2(c) shows the method *computeAndGet()* based on *StampedLock*. It first acquires a write lock (Line 4), and then attempts to convert to a read lock (Lines 7-13) just after the update of the field *length*. To downgrade a lock, the method *tryConvertToReadLock()* of the class *StampedLock* is leveraged to perform the state conversion of the lock *slock*. The variable *stamp* will be updated if the conversion succeeds; otherwise, it can release the write lock and acquire a read lock.

Figure 3 shows the method *getLength()* based on the built-in monitor, ReadWriteLock and *StampedLock*. This method only reads the field *length* and returns it as the returned value.

Figure 3(a) shows that the method *getLength()* only returns the value of the field *length* with the synchronized declaration. When this method is refactored to *ReadWriteLock* by *Relocker* [5], as shown in Figure 3(b), a read lock will be inferred since *Relocker* cannot find any side-effect of this method. Although a read lock is a good choice and introduces more concurrency, this method can be further improved to use an optimistic read lock. We notice that this method has a very short read-only code segment that meets the *StampedLock* ŕs requirement to use the optimistic read lock; this code segment may continue to reduce contention and improve throughput.

Figure 3(c) shows an alternative of the method *getLength()* based on the *StampedLock* to read the field *length*. Instead of acquiring a lock, it first leverages the method *tryOptimisticRead()* to attempt the optimistic read (Line 4), and then reads the field *length* directly (Line 5). Since reading the field in the optimistic mode may be wildly inconsistent, the method *validate()* of the class *StampedLock* is used to check consistency (Line 6). If the current optimistic mode is invalidated by a write operation, it enters the read mode by acquiring a read lock and reads this *length* again (Lines 7-12).

III. REFACTORING FOR STAMPEDLOCK

In this section, we first present an overview of refactoring for *StampedLock*. Then, we discuss the design of the individual components in more detail in Section III-B to III-D.

A. REFACTORING FRAMEWORK

The framework of converting a program based on the synchronized lock into one based on *StampedLock* is presented in Figure 4. The program analysis tool WALA [9] is used to perform analysis based on the WALA intermediate representation (IR). Precondition validation is used to validate whether a synchronized lock can be transformed or not. Reentrance analysis is leveraged to check the reentrance for a synchronized method or block. Precondition checking also attempts to find all the thread communication operations to which *StampedLock* cannot be applied. All locks are located by a visitor pattern analysis. They further undergo a side effect analysis and are converted to write lock, read lock, optimistic read lock, upgrading lock and downgrading lock.

B. PRECONDITION

CLOCK checks two preconditions before refactoring. These preconditions are inherent to how *StampedLock* is used and are not the limitations of our refactoring tool.

1) CONDITIONAL OPERATIONS

The methods *wait()*, *notify()* and *notifyAll()* are used to establish the communication between the threads. However, *StampedLock* does not support these methods. It is mainly because it supports the coordinated usage across multiple lock modes, so it does not directly implement the

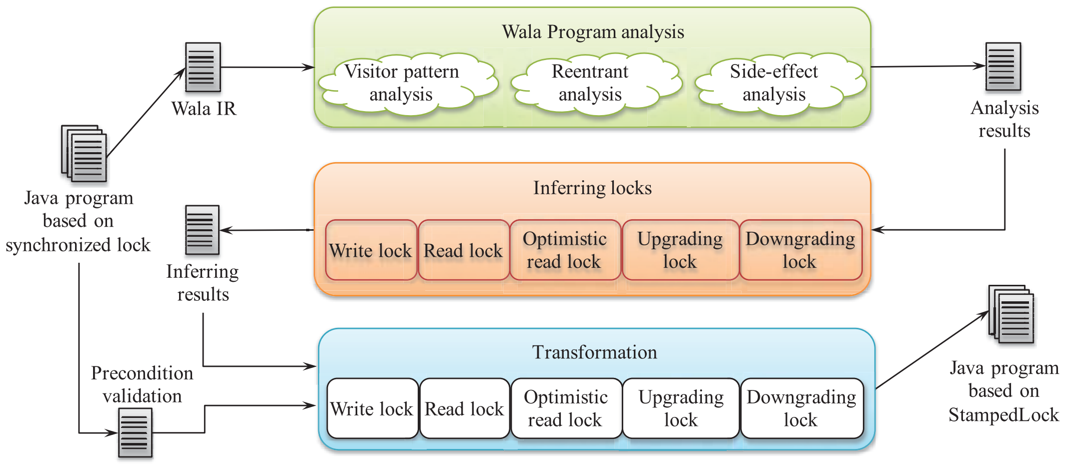


FIGURE 4. **The refactoring framework.**

interface *Lock* such as *ReentrantLock* and *ReentrantReadWriteLock* do. Instead, it provides the methods *asReadLock()*, *asWriteLock()* or *asReadWriteLock()* to be viewed as a read-write lock. Unfortunately, it does not support a condition on the lock returned by invoking these methods and if the *Lock.newCondition()* method is called, an exception *UnsupportedOperationException* will be thrown. Since *StampedLock* does not support conditional operations, CLOCK checks if a method contains them.

2) REENTRANCE ANALYSIS

Wloka *et al.* [10] presented a mostly-automated refactoring that makes programs reentrant by replacing the global state with the thread-local state. They defined the reentrance as "distinct executions of program on distinct inputs cannot affect each other". Different from their work, reentrance in our work means that if a synchronized lock is acquired with a monitor, this lock may be re-acquired with the same monitor. Some locks can be acquired multiple times. For example, *ReentrantReadWriteLock* supports a maximum of 65,535 recursive write locks and 65,535 read locks. Some locks are not reentrant, and *StampedLock* is a typical example. If a lock is not reentrant and a thread tries to acquire the lock held by other threads, the acquisition will not succeed. In other words, a method (or block) that is protected by the *StampedLock* should not call another method (or block) that may try to re-acquire locks with the same monitor of the *StampedLock*.

What if a method body that holds a *StampedLock* acquires this lock again? To answer this question, we should try to understand the nature of the *StampedLock*. Different from the previous locks before JDK1.8, *StampedLock* returns a long integer value *stamp* when acquiring a lock and releases the lock with this *stamp*. If the invoked method re-acquires this lock, the value *stamp* will be updated. As a result, the caller method will never release the lock by using the updated *stamp*. Hence, a deadlock state will occur.

Converting a reentrant lock to a non-reentrant lock will definitely change the behavior of the original program and vice versa. Therefore, our refactoring seems to be impractical. However, we can detect the reentrance and avoid such a transformation wherever the synchronized lock is reentrant. CLOCK will explore the refactoring possibility of those code segments without reentrance.

Figure 5 presents an algorithm of detecting reentrance for both a synchronized method and a synchronized block implemented by *isReentrantForMethod* and *isReentrantfForInstruction* methods, respectively. As the synchronized lock can be used in the declaration of a method or as a block, this algorithm handles the following four situations: (1) the synchronized method calls the synchronized method; (2) the body of the synchronized method involves a synchronized block; (3) a synchronized method is called in a synchronized block; and (4) one synchronized block involves another synchronized block.

Alias analysis is used to analyze monitors that have a different name but actually access the same memory position. For a synchronized method, the monitor is *this* or *A.class* where *A* represents the name of a class. For a synchronized block, the monitor depends on a specific object. The method *isReentrantForMethod* is used to handle both synchronized methods and other non-synchronized methods that contain synchronized blocks. The monitor object of the current method is recorded in *pointerKey* and examines the may alias (Lines 2-5). This method may call itself recursively or other methods, and the algorithm checks all called methods (Lines 6-12). If a method is not synchronized but involves synchronized blocks, it will invoke the method *isReentrantForInstruction* to handle the synchronized blocks (Lines 13-17). The method *isReentrantForInstruction* is used to handle the synchronized block. It obtains the monitor object of the current synchronized instruction and examines the may alias (Lines 25-27). For some situations in which one synchronized block involves another synchronized block,

```
1  boolean isReentranceForMethod(Method mthd, Set pointerKeySet) {

2      if(mthd is modified by synchronized){
3        pointerKey ← the monitor object of the current method
4        if (mayAlias(pointerKeySet, pointerKey))
5          return true;

6        insert pointerKey into pointerKeySet
7        for(each method callee in mthd){
8          if (isReentranceForMethod(callee, pointerKeySet))
9             return true;
10       }
11       remove pointerKey from pointerKeySet
12     }

13     for(each instruction instr in method mthd){
14       if (isReentrantForInstruction(instr, pointerKeySet))
15          return true;
16     }
17     return false;

18 }

19 boolean isReentrantForInstruction (Instruction instr, Set pointerKeySet) {
20    if( instr is a synchronized instruction){
21       if (Instr is a recursively synchronized invocation) {
22          push instrPointerKey into the stack stack;
23          insert instrPointerKey into pointerKeySet;
24       }
25       instrPointerKey ← the monitor object of the current instruction
26       if (mayAlias(pointerKeySet, instrPointerKey))
27           return true;
28        if (Instr is a recursively synchronized invocation) {
29           pop instrPointerKey from the stack stack;
30           remove instrPointerKey from pointerKeySet;
31           if ( instr is a method-call instruction){
32               insert instrPointerKey into pointerKeySet;
33               invokedFunc ← get the method that being invoked;
34               if (isReentranceForMethod(invokedFunc , pointerKeySet))
35                  return true;
36               remove instrPointerKey from pointerKeySet;
37           }
38        }
39    }
40    return false;
41 }
```

FIGURE 5. Algorithm of detecting reentrance for locks.

CLOCK handles the nested synchronized block by a stack (Lines 22 and 29) to ensure that the current *pointKey* is the current monitor object. If a synchronized block invokes a synchronized method, CLOCK calls the *isReentrantForMethod* method to handle it (Lines 31-36).

Note that CLOCK cannot obtain the *pointerKey* for a static synchronized method by WALA [9]. To handle it, CLOCK converts the static synchronized method to a synchronized block with an explicit monitor; then, the *pointerKey* can be obtained.

C. INFERRING LOCKS

CLOCK uses a side-effect analysis to infer the read/write lock, optimistic read lock or upgrading/downgrading lock. *Relocker* [5] presented a side-effect analysis to infer the read/write lock for *ReentrantReadWriteLock*. For each lock, *Relocker* just gives the suggestion of using a read/write lock. Considering our refactoring, *StampedLock* has multiple kinds of lock modes, and the use of read lock modes relies on the associated code sections being side-effect-free. Refactoring for *StampedLock* needs to infer not only the read/write lock but also the upgrade/downgrade lock. The side effect analysis for CLOCK needs to record each read/write operation to facilitate the following inference. If the critical section contains multiple read/write operations, the analysis results will generate a character sequence.

The regular expression is defined for each lock mode to match the character sequence. CLOCK defines five regular expressions for inferring lock modes, where R represents the read operation, W represents the write operation, $A \odot B$ represents A matches B, R^* represents that R repeats for zero or multiple times, and R^+ represents that R repeats once or multiple times.

Regulation 1: CLOCK infers a write lock WL if $WL \odot \{R|W\}^* W \{R|W\}^*$.

Regulation 1 shows that a write lock is inferred if there is at least one write operation in the character sequence. The regulation expression of write lock definitely includes the upgrading/downgrading lock. CLOCK performs a further definition for the upgrading/downgrading lock.

Regulation 2: CLOCK infers an optimistic read lock OL if $OL \odot R_{field}$.

Regulation 2 shows that an optimistic read lock is inferred if there is a read operation to a field of a class.

Regulation 3: CLOCK infers a read lock RL if $(RL \subseteq \neg OL) \cap (RL \odot R^+)$.

Regulation 3 shows that a read lock is inferred if there is at least one read operation and it is not a read operation to the field. Here, $\neg OL$ represents that CLOCK excludes the optimistic read mode out of read mode to avoid the ambiguity of inferring locks.

Regulation 4: CLOCK infers a downgrading lock DL if $(DL \subseteq WL) \cap (DL \odot W^+ R^+)$.

Regulation 4 shows that a downgrading lock is a subset of WL, and write operations are usually followed by one or multiple read operations.

Regulation 5: CLOCK infers an upgrading lock UL if $(UL \subseteq WL) \cap (UL \odot R^+ [\{R|W\}^* W \{R|W\}^*])$.

Regulation 5 shows that a downgrading lock is the subset of *WL* and is inferred if one or multiple read operations follows multiple read and write operations. The '[]' represents the scope of an if-statement in the critical section. We should note that our upgrading lock strategy is only for the situation in which one or multiple read operations are followed by an if-statement that contains write operations. Our strategy for inferring the upgrading lock maybe simple, but most of them happen around the 'if' statement.

D. TRANSFORMATION

CLOCK traverses the abstract syntax tree to locate all synchronized locks and converts the synchronized lock to the corresponding *StampedLock*. In each refactored class, CLOCK imports the *StampedLock* package and defines the *StampedLock*. All *WL*s and *RL*s are put in to the *try...finally...* structure to ensure the release of a lock even if the exception occurs. For *OL*, CLOCK handles the transformation of the optimistic read operation in the same way as the code shown in Figure 3(c). For *DL*, CLOCK downgrades a write lock to a read lock just after the last write operation. For *UL*, CLOCK upgrades a read lock to a write lock just after the 'if' statement.

IV. HANDLING PRACTICAL ISSUES

This section presents several practical issues that we solved during the implementation of CLOCK.

A. EARLY RETURNING

Early returning means that the refactored function ends early. Considering an example similar to Figure 3, if the body of the *getLength()* method in Figure 3(a) only contains one statement '*return length*', when it is refactored analogous to the approach in Figure 3(c), this statement will be placed in Lines 5 and 9. The execution of this method will always end in Line 5 and all the following statements will never be executed.

To handle early returning, we need to create a new local variable with the same type as the variable in the *return* statement. Actually, Figure 3(c) present the possible solution of early returning.

A similar problem can happen if the method *getLength()* contains an output statement as follows.

```
public synchronized void getLength() {
    System.out.println("length=" + length);
}
```

When refactoring the above code segment, we cannot replace the statement in Lines 5 and 9 of Figure 3(c) with this output statement. Otherwise, the value *length* will be printed twice if the validation fails. Creating a new local variable can solve this problem as follows.

```
public void getLength() {
    int temp;
    long stamp = slock.tryOptimisticRead();
    temp = length;
    if(!slock.validate(stamp)){
        stamp=slock.readLock();
        try{
            temp = length;
            System.out.println("length=" + temp);
        } finally {
            slock.unlockRead(stamp);
        }
    } else {
        System.out.println("length=" + temp);
    }
}
```

B. THE CHANGE OF THE VARIABLE SCOPE

For some variables defined in the critical section, CLOCK may change the variable scope because CLOCK will use the *try...finally...* structure and move the original critical section into the *try* block. As a result, the variable scope may become small. To solve this problem, CLOCK checks these definitions of the variable and allows them to be defined out of the *try* statement.

C. ESCAPING

Escaping analysis is an approach for determining the dynamic scope of a pointer. A pointer to a variable that defined in a thread can escape into other threads. Considering *StampedLock*, it returns a *stamp* value after acquiring the lock, and leverages this *stamp* value to release the lock. CLOCK should make sure that the variable *stamp* will not escape out of the scope of the current thread. If the pointer to the variable *stamp* escapes to the other threads, its value will likely be changed, making the lock unreleased and finally leading to deadlock.

When converting from the synchronized lock to the *StampedLock* via CLOCK, the *stamp* is a local variable and the scope is within a method. The local variable *stamp* cannot be returned and passed to another method. Hence, CLOCK ensures the variable *stamp* will not escape.

D. AVOIDING SWITCHING BETWEEN UPGRADING AND DOWNGRADING LOCKS FREQUENTLY

The upgrading/downgrading lock enables concurrent programs to upgrade a read lock to a write lock or to downgrade a write lock to a read lock. If a code segment contains more than one upgrading/downgrading operation, lock mode will be switched frequently so that acquiring and releasing lock operations dominate the execution time, which will definitely decrease the performance. CLOCK uses a write lock instead of multiple upgrading/downgrading locks to avoid such a frequent switching.

V. IMPLEMENTATION

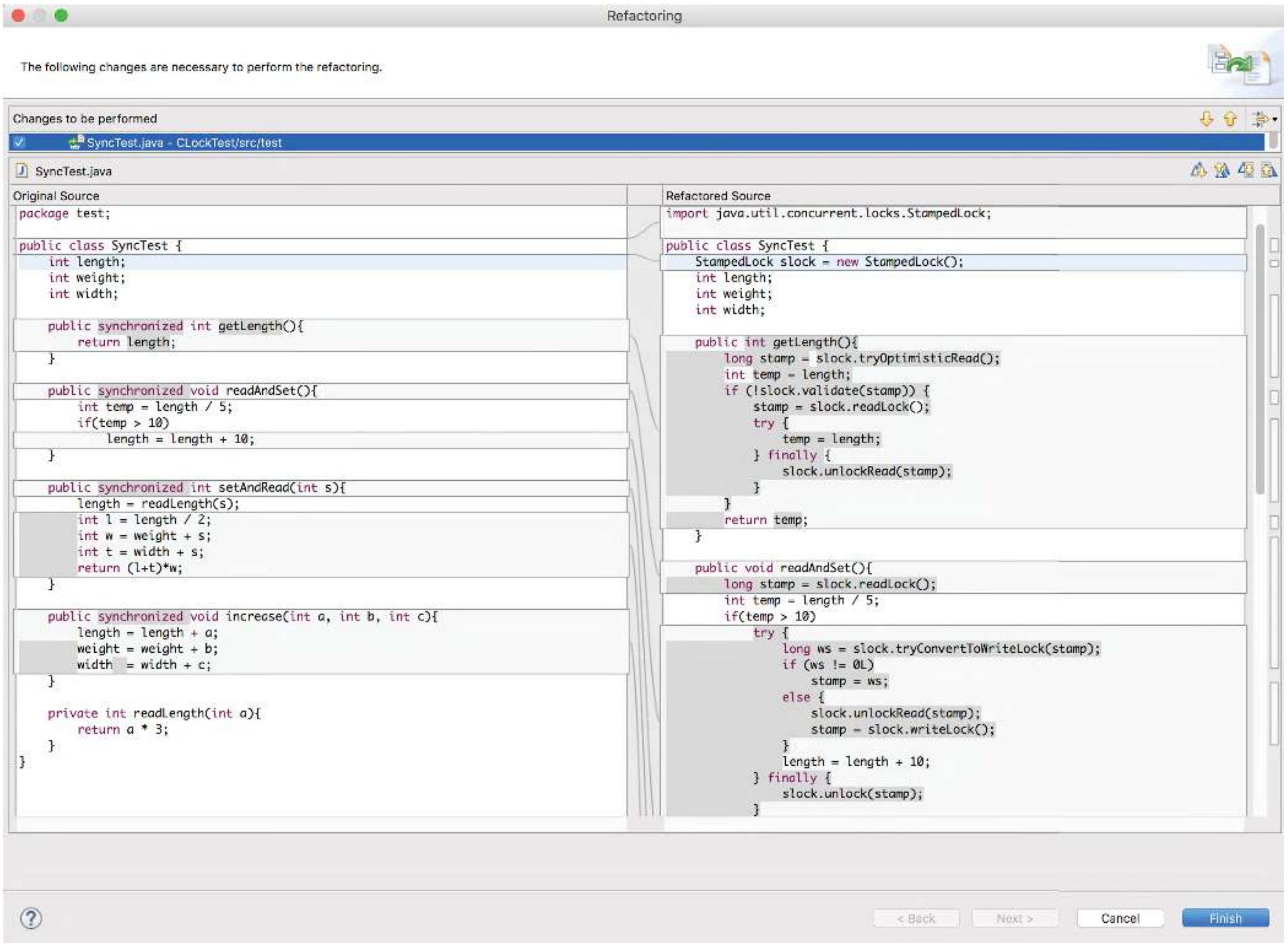
We implement our refactoring in a prototype tool called CLOCK as an extension to Eclipse IDE. The screenshot of the CLOCK implementation is presented in Figure 6. The left-hand side of Figure 6 presents a customized class *SyncTest* that includes three fields and five methods, while the right-hand side of Figure 6 shows the refactored results by using *StampedLock*.

VI. EVALUATION

This section first introduces the experimental setup and benchmarks, and then presents the research questions and illustrates the experimental results.

A. EXPERIMENTAL SETUP AND BENCHMARKS

All experiments are conducted on a MacBook Pro with a 2.5GHz Intel Core i7 CPU, 16GB RAM, and 6MB cache.



The following changes are necessary to perform the refactoring.

Original Source

```java
package test;

public class SyncTest {
    int length;
    int weight;
    int width;

    public synchronized int getLength(){
        return length;
    }

    public synchronized void readAndSet(){
        int temp = length / 5;
        if(temp > 10)
            length = length + 10;
    }

    public synchronized int setAndRead(int s){
        length = readLength(s);
        int l = length / 2;
        int w = weight + s;
        int t = width + s;
        return (l+t)*w;
    }

    public synchronized void increase(int a, int b, int c){
        length = length + a;
        weight = weight + b;
        width  = width + c;
    }

    private int readLength(int a){
        return a * 3;
    }
}
```

Refactored Source

```java
import java.util.concurrent.locks.StampedLock;

public class SyncTest {
    StampedLock slock = new StampedLock();
    int length;
    int weight;
    int width;

    public int getLength(){
        long stamp = slock.tryOptimisticRead();
        int temp = length;
        if (!slock.validate(stamp)) {
            stamp = slock.readLock();
            try {
                temp = length;
            } finally {
                slock.unlockRead(stamp);
            }
        }
        return temp;
    }

    public void readAndSet(){
        long stamp = slock.readLock();
        int temp = length / 5;
        if(temp > 10)
            try {
                long ws = slock.tryConvertToWriteLock(stamp);
                if (ws != 0L)
                    stamp = ws;
                else {
                    slock.unlockRead(stamp);
                    stamp = slock.writeLock();
                }
                length = length + 10;
            } finally {
                slock.unlock(stamp);
            }
```

FIGURE 6. CLOCK converts synchronized locks (left-hand side) into StampedLocks (right-hand side).

The machine runs OS X EI Capitan and has JDK 1.8.0_25, Eclipse 4.4.1 and WALA 1.4.2 installed.

To evaluate the usefulness of CLOCK, we run it on 3 projects including the SPECjbb2005 [11] benchmark and two real-world applications: Xalan [12] and FOP [13]. SPECjbb2005 was developed by the Standard Performance Evaluation Corporation as a benchmark for evaluating the performance of server side Java by emulating a 3-tier system with an emphasis on the middle tier. It provides an enhanced workload to reflect realistic applications. Xalan is the open source software library from the Apache project that can transform XML documents into HTML, text or other XML document types using the XSLT standard stylesheet. We use its Java version 2.7.2. A formatting objects processor (FOP) is also part of the Apache project, reads a formatting object tree and renders the resulting pages to a specific output. Its version is 2.3. For each benchmark, we apply CLOCK to all synchronized methods and blocks. CLOCK checks the preconditions and makes the transformation when they pass the precondition validation.

B. RESEARCH QUESTIONS

We evaluate the effectiveness of CLOCK by answering the following questions:

- RQ1: How applicable is the refactoring? In other words, how many synchronized methods and blocks can meet the refactoring precondition?
- RQ2: Are these refactorings correct?
- RQ3: Can CLOCK save developer effort when refactoring?

We answer RQ1 by counting how many code fragments meet the refactoring preconditions and thus are refactored by CLOCK. We also report the number of times that the precondition failed. RQ2 is answered by inspecting these changes and reporting the possible inference. To measure how many efforts a developer would spend on manually refactoring, RQ3 is answered by reporting the number of files [14] modified by the refactoring. We also report the number of modified SLOC. SLOC is generated using SLOCCount [15]. These numbers approximately estimate the programmer effort that is saved when refactoring with CLOCK.

Benchmark	Original benchmark			Refactored benchmark									
	#SM	#SB	SLOC	#Write locks	#Read locks	#Optimistic read locks	#Upgrading locks	#Downgrading locks	#Classes that can be refactored	#Classes that can't be refactored	#SM that can't be refactored	#SB that can't be refactored	SLOC
SPECjbb2005	168	22	12544	65	14	36	0	0	19	4	57	18	13083
Xalan	51	31	171522	43	3	2	8	0	29	4	20	6	171843
FOP	25	7	211750	15	6	0	8	1	18	1	2	0	211974
Total	244	60	395816	123	27	34	16	1	66	9	79	24	396900

#SM: number of synchronized methods #SB: number of synchronized blocks SLOC: source lines of code

FIGURE 7. **Experimental results for CLOCK.**

C. RESULTS

Figure 7 tabulates the results for CLOCK. For each original benchmark, it shows the number of synchronized methods and blocks as well as SLOC. For each refactored benchmark, we demonstrate how many write locks, read locks, optimistic read locks, upgrading locks and downgrading locks are inferred and refactored. We also report how many synchronized methods and blocks cannot be refactored.

1) RESULTS FOR RQ1

Column 10 in Figure 7 shows the number of classes that can be refactored, while column 11 shows the number of classes that cannot be refactored. The number of synchronized methods and synchronized blocks that cannot be refactored is shown in columns 12 and 13. A total of 79 synchronized methods and 24 synchronized blocks fail the validation of preconditions. However, these synchronized locks are spread across a small range of classes.

For SPECjbb2005, the original benchmark contains 168 synchronized methods and 22 synchronized blocks that are widely spread across 23 files. Moreover, 61% of synchronized methods and blocks meet the refactoring preconditions. There are 4 classes (including *Company*, *DeliveryTransaction*, *TimerData*, and *Warehouse*) that are not transformed because they fail the validation of preconditions. In addition, 57 synchronized methods and 18 synchronized blocks cannot be refactored. Two reasons lead to the failure.

- The most commonly failed preconditions are caused by lock reentrance. Both direct reentrance and indirect reentrance are found by CLOCK. For example, the *primeWithDummyData()* method of the *Company* class called the *loadInitialOrders()* method in this class. Both of them are synchronized methods and use the instance of the *Company* class as the monitor. It is direct reentrance. Indirect reentrance also exists. For example, the *process()* method of the *DeliveryTransaction()* class calls the *handleDelivery()* method of the *DeliveryHandler* class, and then the *handleDelivery()* method calls the *display()* method of the *DeliveryTransaction()* class. Both *process()* and *display()* are synchronized methods with the same monitor.
- The remainder of failed preconditions is caused by thread communication operations that are included in the synchronized blocks. CLOCK cannot convert them because *StampedLock* does not support the conditional operation.

For Xalan, the original project has 51 synchronized methods and 31 synchronized blocks. Moreover, 68% of synchronized methods and blocks meet the refactoring precondition. However, there are 20 synchronized methods and 6 synchronized blocks that cannot be refactored. Almost all failed preconditions for synchronized blocks are caused by thread communication operations that *StampedLock* cannot handle. For synchronized methods that fail the validation of preconditions, 15 synchronized methods are caused by the reentrance and 5 of them are caused by thread communication operations. These synchronized locks that cannot be refactored are spread across only 4 classes.

For FOP, the original project has 25 synchronized methods and 7 synchronized blocks. Furthermore, 94% of synchronized locks meet the refactoring precondition. Only 2 synchronized methods cannot be refactored because they are reentrant.

The experimental results show that some synchronized locks cannot be refactored due to the limitation of *StampedLock*. However, we notice that they are only spread across a small range of 9 classes. When a method in a class fails the validation due to reentrance, this leads to the failure of validating other methods in this class. From the perspective of the number of classes, CLOCK still has a high level of applicability.

2) RESULTS FOR RQ2

A total of 123 write locks, 27 read locks, 34 optimistic read locks, 16 upgrading locks and 1 downgrading lock are inferred by CLOCK. We check each transformation manually and find that CLOCK transforms all synchronized locks and does not miss any refactorings. To check if these refactorings are correct or not, we manually inspect all the refactored locks to determine 1) if a correct kind of lock is inferred or not; 2) if a lock is inserted into a correct position or not; 3) if a lock structure is used correctly or not; and 4) if the critical section is protected safely or not.

During the inspection, we find that each critical section has been inferred with the kind of lock according to the lock inference strategy (see Section III-C) and almost all of them are accurate. However, we also find that the inferred locks for some critical sections can be improved by using other locks. These cases are reported as follows. We should note that these cases are related to the kinds of locks that are used and are not related to the correctness of programs and validity of CLOCK.

- The critical section that is inferred to use a write lock can be improved by using an upgrading/downgrading lock. For example, method *removeOldNewOrders()* of class *District* in the SPECjbb2005 benchmark first contains a read operation, then a write operation, and finally all read operations, which may use a downgrading lock. However, a write lock is inferred according to our inference strategy.
- The critical section that is inferred to use a read lock can be improved by using an optimistic read lock. For example, method *display()* of class *District* in the SPECjbb2005 benchmark reads three fields. According to our lock inference strategy, reading just one field recommends to use an optimistic read lock. This method does not meet the reference strategy for an optimistic read lock. Therefore, a read lock is inferred for this method.

We do not find any refactorings that change lock semantics. More precisely, all locks are inserted into the position where they should be and thus all critical sections are protected safely by locks. We also inspect the lock structures and find that all of them are used correctly. For example, for read/write lock, all acquire operations are inserted before the critical sections and all release operations are put into the 'finally' blocks. As for optimistic read lock, the direct read operations are validated. If they fail the validation, a read lock is used to protect reading again.

To make sure that all benchmarks work well, we run the refactored programs. For the SPECjbb2005 benchmark, we run it against the number of threads (1 thread to a maximum of 16 threads). It shows the score and heap memory usage for each run if there is no error. For Xalan and FOP, we run them by using the samples published with the source code to transform or render an XML document. We find that they all run smoothly without reporting any errors.

3) RESULTS FOR RQ3

Measuring the developer effort in terms of a precise evaluation is truly difficult. Ideally, we would have observed developers while they refactor and determined how much time they spend. However, given the differences in familiarity with concurrent programming for different developers, the refactoring time may vary. To approximately estimate the effort of manual refactoring, we select three graduate students who are familiar with the *StampedLock* and let them transform each project manually. As a result, they take 5 hours on average to accomplish the transformation.

We count the number of synchronized locks together with the number of code changes. These figures represent that a developer would have spent time in searching for synchronized locks and transformed the code manually. In total, all benchmarks have 244 synchronized methods and 60 synchronized blocks that are spread across 395KSLOC. A total of 66 classes are modified by applying the refactoring, with an average of 22 classes per project. The refactorings modify 1084 SLOC, with an average of 361 SLOC per project. When refactoring the SPECjbb2005 benchmark, developers need to search for synchronized locks in 23 files out of 33 files in total, and eventually 190 refactorings are found; then, developers determine which lock should be used and perform transformations. The synchronized locks are clustered in the SPECjbb2005 benchmark so that developers can find these locks quickly. However, the situation is quite different for the other two benchmarks, in which the refactorings are not strongly clustered. If developers transform Xalan manually, they need to search 920 files to find synchronized locks existed only in 29 files. It is labor-intensive to search in such a large amount of files to find a small amount of locks and convert them to *StampedLock*. Eventually, 82 locks are required to be transformed manually by developers, and 321 SLOC are modified. For FOP, 32 synchronized locks are spread across 2004 Java files. If a developer refactored manually, he/she would have had to jump across many files. Finally, only 224 SLOC are modified.

By contrast, our tool is fully automatic. For SPECjbb2005, a total of 19 classes are modified by applying the refactoring. CLOCK infers 65 write locks, 14 read locks, and 36 optimistic read locks. For Xalan, CLOCK infers 43 write locks, 3 read locks, 2 optimistic read locks and 8 upgrading locks. We check 8 upgrading locks manually and find that most of them are similar to our proposed motivation presented in Figure 1. For FOP, CLOCK infers 15 write locks, 6 read locks, 8 upgrading locks, and 1 downgrading lock. The downgrading lock occurs in the *generateNewID* method of the *ActionSet* class where a write operation is followed by several read operations; this pattern is similar to our proposed motivation in Figure 2. It takes no more than thirty seconds per project. These results show that CLOCK can save considerable developer efforts.

VII. RELATED WORKS

In this section, we first investigate programming libraries and tools that support the upgrading/downgrading lock, and then present refactoring for locks. Finally, we examine the works on refactoring for different synchronization mechanisms.

A. PROGRAMMING TOOLS THAT SUPPORT THE UPGRADING/DOWNGRADING LOCK

Many programming libraries and frameworks have provided mechanisms to support upgrading/downgrading lock operations. Early after the JDK 1.5 proposal, the

ReentrantReadWriteLock class allowed downgrading from a write lock to a read lock. However, upgrading from a read lock to a write lock is not feasible. Importantly, since JDK 1.8, *StampedLock* has supported both the upgrading and downgrading lock. Furthermore, the optimistic read lock is recommended for all new development of accessing fields.

The .NET Framework provides the *ReaderWriterLockSlim* [16] class to support three modes: read mode, write mode, and upgradable read mode. *ReaderWriterLockSlim* is similar to *ReaderWriterLock*, but the performance of *ReaderWriterLockSlim* is significantly better than that of *ReaderWriterLock*. An upgradable mode is intended for cases where a thread usually reads from the protected resource but might need to write to it if some condition is met. *ReaderWriterLockSlim* has simplified rules for the upgrading and downgrading lock state. However, the .NET framework does not support the automated refactoring for these read and write locks.

Intel Threading Building Blocks(TBB) provide methods *downgrade_to_reader* and *upgrade_to_writer* to support the downgrading/upgrading lock [17]. However, TBB does not provide refactoring for these methods.

B. REFACTORING FOR LOCKS

Many previous works of concurrency-oriented refactoring focused on how to convert locks.

McCloskey *et al.* [18] presented *Autolocker* to automatically convert the pessimistic atomic section into lock-based code. *Autolocker* retained many of the advantages of optimistic atomic sections and reduced the most burdens of lock-based programming. In addition, they allowed programmers to extract more parallelism through fine-grained locking.

Schäfer *et al.* [5] presented algorithms to convert built-in monitor locks into *ReentrantLock*s and *ReentrantReadWriteLock*s. They also claimed that their future works would involve helping developers to safely downgrade write locks to read locks. Inspired by their work, our work presented an algorithm to upgrade read locks to write locks, downgrade write locks to read locks and use optimistic read locks.

Tao and Qian [6] proposed an automated refactoring approach for Java concurrent programs based on synchronization requirements. Their work had the ability to find the refactoring opportunities for splitting locks, splitting the critical section and converting it to the atomic section. Zhang *et al.* [19] presented a refactoring approach for lock based on bytecode transformation.

Some commercial refactoring tools, such as concurrency-oriented refactoring for JDT [7] and LockSmith [8], have been integrated into IntelliJ IDEA and Eclipse respectively. Both of them can split and merge locks, convert among locks, and make the field atomic.

Although many refactorings for locks had been performed, most previous works mainly concentrated on read/write locks and few works focused on the upgrading/downgrading lock or optimistic read lock.

C. REFACTORING FOR DIFFERENT SYNCHRONIZATION MECHANISMS

There has been considerable interest in refactoring programs among different synchronization mechanisms.

Deng *et al.* [20] proposed a tool *SyncGen* to automatically synthesize complex synchronization implementations from formal high-level specifications. *SyncGen* also provided checkable redundancy for verifying the correctness of synthesized implementations, and exploited synchronization specifications for state-space reduction of general correctness properties.

The refactoring tool *CONCURRENCER* of Dig *et al.* [21] aims to convert synchronized locks to atomic blocks. Programmers can replace all *int* field accesses with calls of *AtomicInteger* thread-safe APIs. Regarding the difference between locks and atomic blocks, *CONCURRENCER* can only transform those synchronized blocks that contain one-field accesses.

Ishizaki *et al.* [22] proposed a refactoring approach to support atomic refactoring. Their tool transforms more refactoring cases for the *java.util.concurrent.Atomic.AtomicInteger* class.

Zhang [23] proposed an aspect-oriented synchronization library *FlexSync*. *FlexSync* enabled programmers to choose synchronization control among lock, atomic block and STM only by adding aspect-oriented annotation. Programmers could evaluate the performance of Java programs using different synchronization control mechanisms. *FlexSync* supported complex Java systems simultaneously working with multiple synchronization mechanisms without any code changes.

VIII. CONCLUSION

StampedLock has been designed to enhance the synchronization control for concurrent programs by providing the upgrading lock and optimistic read lock, which are Java locks that were previous unavailable. This paper first illustrates several motivations that might improve the design by using *StampedLock*, and then presents the analysis and algorithms of refactoring that enable Java developers to convert the synchronized lock to *StampedLock*. CLOCK is implemented as the Eclipse plugin and evaluated with three large applications. The evaluation shows that a total of 66 classes are modified by searching approximately 395KSLOC and applying the refactoring, with an average of 22 classes per benchmark. Experimental results provide confidence that the proposed algorithms and implementation can help the developer with refactoring and save developer effort.

A threat to validity of our evaluation is that the benchmark programs may not be representative of all programs. Considering that different kinds of programs have different characteristics, they may exhibit all kinds of lock behaviors.

Although *StampedLock* provides promising lock operations, it introduces some obstacles (e.g. unsupported reentrancy and conditional operations) for refactoring. If JDK were to provide an advanced lock (similar to *StampedLock*) together with support of reentrance and condition operations, CLOCK could achieve a higher refactoring success rate.

Future works include our endeavors to find more refactoring patterns for the upgrading/downgrading lock and to work on how to ensure consistency when converting locks. Although carefully developed and tested, CLOCK is a prototype tool that may contain bugs and still needs more efforts to make it sufficiently mature. However, our tool can still be applied in several applications. We will continuously improve our tool to ensure its correctness. Furthermore, we will work on the recommend approach for locks. Importantly, it will be constructive if a tool can recommend the best lock in advance rather than resort to refactoring.

ACKNOWLEDGMENT

The authors would like to thank the insightful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] E. Silvestri, S. Economo, P. Di Sanzo, A. Pellegrini, and F. Quaglia, "Preemptive software transactional memory," in *Proc. IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, May 2017, pp. 294–303.

[2] A. Khyzha, H. Attiya, A. Gotsman, and N. Rinetzky, "Safe privatization in transactional memory," *ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, vol. 53, no. 1, pp. 233–245, 2018.

[3] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," *J. Parallel Distrib. Comput.*, vol. 70, no. 1, pp. 1–12, 2010.

[4] H. M. Kabutz. (2008). *Starvation With ReadWriteLocks*. [Online]. Available: https://www.javaspecialists.eu/archive/Issue165.html

[5] M. Schafer, M. Sridharan, J. Dolby, and F. Tip, "Refactoring java programs for flexible locking," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 71–80.

[6] B. Tao and J. Qian, "Refactoring java concurrent programs based on synchronization requirement analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep./Oct. 2014, pp. 361–370.

[7] K. Ahti. (2010). *Concurrency-Related-Refactorings-for-JDT*. [Online]. Available: https://wiki.eclipse.org/Concurrency-related-refactorings-for-JDT

[8] Sixth and Red River Software. (2008). *LockSmith: Concurrency-Oriented Refactorings for IntelliJ IDEA*. [Online]. Available: https://intellij-support.jetbrains.com/hc/en-us/community/posts/206761105–Ann-LockSmith-concurrency-oriented-refactorings-for-IntelliJ-IDEA

[9] (2018). *The T. J. Watson Libraries for Analysis*. [Online]. Available: https://github.com/wala/WALA

[10] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for reentrancy," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 173–182.

[11] SPEC. (2013). *SPEC JBB2005*. [Online]. Available: http://www.spec.org/jbb2005/

[12] A. S. Foundation. (2019). *Xalan-Java Version 2.7.2*. [Online]. Available: https://xalan.apache.org/xalan-j/index.html

[13] Apache. (2019). *The Apache FOP Project*. [Online]. Available: https://xmlgraphics.apache.org/fop/

[14] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 543–553.

[15] D. A. Wheeler. (2018). *SLOCCount*. [Online]. Available: https://dwheeler.com/sloccount/

[16] Microsoft. (2018). *ReaderWriterLockSlim Class*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.threading.readerwriterlockslim?view=netframework-4.7.2

[17] Intel. (2018). *Intel Threading Building Blocks Documentation*. [Online]. Available: https://software.intel.com/en-us/node/506088

[18] B. McCloskey, F. Zhou, D. Gay, and E. Brewer, "AutoLocker: Synchronization inference for atomic sections," in *Proc. 33rd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2006, pp. 346–358.

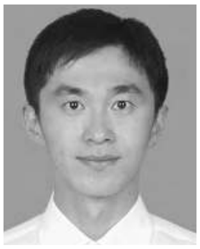
[19] Y. Zhang, S. Shao, H. Liu, J. Qiu, D. Zhang, and G. Zhang, "Refactoring java programs for customizable locks based on bytecode transformation," *IEEE Access*, vol. 7, pp. 66292–66303, 2019.

[20] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno, "SyncGen: An aspect-oriented framework for synchronization," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst. (TACAS)*, 2004, pp. 158–162.

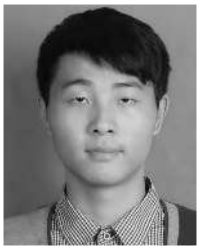
[21] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 397–407.

[22] K. Ishizaki, S. Daijavad, and T. Nakatani, "Refactoring java programs using concurrent libraries," in *Proc. Workshop Parallel Distrib. Syst.. Test., Anal., Debugging*, 2011, pp. 35–44.

[23] C. Zhang, "FlexSync: An aspect-oriented approach to Java synchronization," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 375–385.



YANG ZHANG received the Ph.D degree from the School of Computer, Beijing Institute of Technology. He is currently an Associate Professor with the School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include parallel programming model and software refactoring for parallelism.



SHICHENG DONG is currently pursuing the master's degree with the Hebei University of Science and Technology. His research interests include parallel programming and software refactoring for parallelism.

XIANGYU ZHANG received the Ph.D. degree from the Computer Science Department, The University of Arizona. He is currently a Professor with the Department of Computer Science, Purdue University. His research interests include program analysis, security, deep learning security, dependability, and interpretability.

HUAN LIU is currently pursuing the master's degree with the Hebei University of Science and Technology. Her research interests include parallel programming and software refactoring for parallelism.

DONGWEN ZHANG received the Ph.D. degree from the Beijing Institute of Technology. She is currently a Professor with the School of Information Science and Engineering, Hebei University of Science and Technology. Her research interests include parallel programming model and software refactoring for parallelism.