applied sciences

MDPI

Article

An Automated Refactoring Approach to Improve IoT Software Quality

Yang Zhang [1], Shuai Shao [1], Minghan Ji [1], Jing Qiu [2,*], Zhihong Tian [2] and Xiaojiang Du [3] and Mohsen Guizani [4]

[1] School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050000, China; zhangyang@hebust.edu.cn (Y.Z.); shao724854691@163.com (S.S.); jiminghan123@163.com (M.J.)
[2] Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510006, China; tianzhihong@gzhu.edu.cn
[3] Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA; dxj@ieee.org
[4] Department of Computer Science and Engineering, Qatar University, Doha 2713, Qatar; mguizani@gmail.com
* Correspondence: qiujing@gzhu.edu.cn

Received: 21 October 2019; Accepted: 1 January 2020; Published: 6 January 2020

Abstract: Internet of Things (IoT) software should provide good support for IoT devices as IoT devices are growing in quantity and complexity. Communication between IoT devices is largely realized in a concurrent way. How to ensure the correctness of concurrent access becomes a big challenge to IoT software development. This paper proposes a general refactoring framework for fine-grained read–write locking and implements an automatic refactoring tool to help developers convert built-in monitors into fine-grained *ReentrantReadWriteLocks*. Several program analysis techniques, such as visitor pattern analysis, alias analysis, and side-effect analysis, are used to assist with refactoring. Our tool is tested by several real-world applications including *HSQLDB*, *Cassandra*, *JGroups*, *Freedomotic*, and *MINA*. A total of 1072 built-in monitors are refactored into *ReentrantReadWriteLocks*. The experiments revealed that our tool can help developers with refactoring for *ReentrantReadWriteLocks* and save their time and energy.

Keywords: refactoring; IoT software; synchronization; software quality; concurrency

1. Introduction

The exponential growth of IoT devices is changing our world [1,2]. According to a recent Gartner report [3], 8.4 billion devices including smart phones, tablets and laptops will be connected by 2020, and this number is expected to grow up to 20.4 billion by 2022. To enable smooth interaction between these IoT devices, IoT software should provide good support. [4,5].

Communication between IoT devices is primarily realized in a concurrent way. How to ensure the correctness of concurrent access becomes a major challenge in IoT software development [6,7]. Java has become one of the most popular programming languages for IoT software development because of its capacity to handle concurrency-related problems [8]. Java virtual machine (JVM) provides support for Java-based IoT applications running almost on any chip. Java provides support for IoT software in different aspects, such as cloud computing, big data, sensors, and M2M computing. Java's ability of combining different devices makes it a good choice for development of IoT applications.

Writing a high-quality concurrent program is still challenging. Developers usually employ coarse-grained locks which introduce lock contention and decrease performance.

However, fine-grained locking is notoriously difficult. Usually, it makes programs more complicated, error-prone, and incurs other problems. Sometimes, improper use of fine-grained locking will increase the overhead of lock operations.

Early works on optimization of concurrent programs focused on the compiler level [9,10]. However, the compiler optimization is complicated and subject to limitation of many factors. Some researchers proposed methods, such as lock splitting [11] and class splitting [12], to reduce lock contention, and a method of merging synchronized blocks to reduce lock operation overhead [13]. Max et al. [14] have proposed an algorithm of refactoring for *ReentrantLocks* and *ReentrantReadWriteLocks*, as well as a refactoring tool Relocker. But Relocker does not support fine-grained refactoring. Though this tool allows automation to some extent, it relies heavily on manual selection of codes to refactor. This paper aims to realize fine-grained lock refactoring via lock splitting and lock degrading of *ReentrantReadWriteLocks*, and implements an automatic tool.

Our refactoring method for fine-grained *ReentrantReadWriteLocks* is designed based on static program analysis. Our method identifies all built-in monitors in a program through visitor pattern analysis. The problem of alias with the same monitor object is resolved by alias analysis. The read and write patterns of critical section are analyzed by using effect side analysis. Our refactoring tool works on the source level to help developers implement the automated refactoring from coarse-grained locks to fine-grained *ReentrantReadWriteLocks*. Our tool is tested by five real-world applications, and a total of 1072 built-in monitors have been refactored.

The main contributions of this paper include the following.

- We developed algorithms that could convert built-in monitor locks to fine-grained *ReentrantReadWriteLocks*.
- We developed an automated refactoring tool implemented as Eclipse plugins.
- We evaluated our tool on several real-world applications.

The remainder of this paper is organized as follows. The related works are examined in Section 2. Some advantages of the *ReentrantReadWriteLock* over built-in monitor locks are presented in Section 3. Section 4 demonstrates our refactoring framework, and our refactoring and analysis algorithm design. Some practical problems are discussed in Section 5. Section 6 presents the evaluation of our proposed tool on a set of Java applications, and conclusions and future works are presented in Section 7.

2. Related Work

In the early study of lock-oriented refactoring, Aldrich et al. [9] and Bogda et al. [10] focused on eliminating unnecessary synchronization, but it was complicated on the compiler level and was limited by many factors. Tao et al. [11] proposed a method of lock splitting based on synchronization requirement analysis. Diniz et al. [13] reduced the overhead by coarsening the granularity at which the computation locks objects. Schäfer et al. [14] designed a refactoring tool *Relocker* to convert built-in monitors to *ReentrantLocks* and *ReentrantReadWriteLocks*. Inspired by his work, Zhang et al. [15] worked on refactoring from a built-in monitor to a StampedLock by lock downgrading/upgrading and optimistic synchronization.

Bavarsad et al. [16] proposed a way to overcome the overhead of the global clock for Software Transactional Memory (STM) by two optimization techniques. The first was read–write lock allocation (RWLA), which could only improve the performance of the STM if the transaction committed successfully. However, if conflicts occurred frequently, RWLA would increase the abort cost and reduce performance; the second optimization technique was a dynamic selection baseline scheme or adaptive technique to reduce the abort cost of RWLA.

Emmi et al. [17] proposed an automatic lock allocation technique to infer the location of a lock in a program and ensure that the lock was correct and avoid deadlocks. Kawachiya et al. [18] proposed a lock retention algorithm that allowed a lock to be retained by a thread. When a thread tried to acquire a lock operation, if the thread retained the lock, it would not have to perform an atomic operation to

get the lock; otherwise, the thread would use the traditional method to obtain a lock. Hofer et al. [19] came up with a new method for analyzing lock contention in Java applications by tracking locking events in Java virtual machines. Their method detected not only when threads were blocked on locks, but also when other threads held the lock to block it and recorded their call chains. This method could reveal the causes for lock contention and identify the performance bottlenecks of locks. Our previous research implemented an automatic refactoring tool to convert built-in monitors to *StampedLocks* [15], and refactored Java programs for customized locks [20].

For the software refactoring tools, Dig et al. probed into concurrent refactoring. They proposed a software parallelization refactoring tool, CONCURRENCER [21], which could refactor serial Java codes into parallel Java codes. By refactoring a serial program into a re-entrant parallel program using the java.util.concurrent library, they converted a thread into a Fork/Join framework, converted int. into *AtomicIntegers*, and converted *HashMaps* into *ConcurrentHashMaps*, making data access thread-safe. Tip et al. focused on the validation of software correctness in the early stages of refactoring and designed a refactoring tool *Reentrancer* [22] to make programs reentrant by transforming a sequence program into a reentrant program.

The impact of IoT is worldwide [23,24]. Refactoring can improve the quality of software, but in the meantime incurs security risks [25,26]. Some researchers have paid attention to the security of IoT software [27,28].

3. Motivation

In this section, we first introduce the background of the *ReentrantReadWriteLock* and present some possible application scenarios of the *ReentrantReadWriteLock*. Also, the performance of the *synchronized* lock and the *ReentrantReadWriteLock* is compared.

3.1. Background

The *ReentrantReadWriteLock* [29] is a locking mechanism introduced in JDK 1.5. It maintains a pair of associated locks, read locks, and write locks. As long as there is no write thread, the read lock may be held simultaneously by multiple read threads. The write lock is an exclusive lock and can be held only by one thread at a time.

The *ReentrantReadWriteLock* supports lock downgrading, which means that a current thread can acquire a read lock while holding a write lock, and then release the write lock. Acquiring the read lock and then releasing the write lock is to ensure the visibility of the data. However, a *ReentrantReadWriteLock* does not support upgrading from a read lock to a write lock, the purpose of which is also to ensure data visibility. If the read lock is acquired by multiple threads, any thread among them can successfully acquire the write lock and update the data, but its update is not visible to other threads that have the read lock.

The *ReentrantReadWriteLock* enables more concurrency when accessing shared data. In theory, the performance of using *ReentrantReadWriteLocks* would be significantly better than using mutually exclusive locks. However, in practice, the performance also depends on the concurrent processing power of multi-core processors and access patterns to shared data.

The standard library *java.util.concurrent* [30] has provided classes and interfaces to enable flexible usage of locks, such as *ReentrantLocks*, *ReentrantReadWriteLocks*, and *StampedLocks*. These will allow the program to run with a fine-grained lock. Nevertheless, Pinto et al. [31], after analyzing 2227 Java projects with concurrent structures on SourceForge.net, concluded that the Java concurrency library had not been used sufficiently and only ~23% of Java projects with concurrent programming structures had used it.

3.2. Motivating Example

Figure 1 presents the use of two different lock structures to implement the insert and inquire operations on the database. The program of Figure 1a is implemented by *synchronized*, where *inquire()*

is a query operation for the database and *insert()* is an insert operation. The program in Figure 1b is implemented by *ReentrantReadWriteLocks*, where the query operation *inquire()* is a read operation, so the read lock is used. The insertion operation *insert()* is a write operation, so the write lock is used. Note that when using *ReentrantReadWriteLocks* for synchronization, the *unlock()* command must be called to release the read lock or the write lock after the operation that requires synchronization. A try-finally construct is usually used when the *ReentrantReadWriteLock* is used, and the operation of releasing the lock is placed in the final block to ensure that the lock is always released to avoid deadlocks.

```
class SynDB{
   ...
  public synchronized Object inquire(){
     //query data from the database
  }
  public synchronized void insert(Object data){
     //insert data into the database
  }
}
```

(a) Implemented with synchronized

```
class SynDB{
  ...
 ReentrantReadWriteLock rwl=new ReentrantReadWriteLock();
 public Object inquire(){
      rwl.readLock().lock();
      try{
         //query data from the database
      }finally{
         rwl.readLock().unlock();
      }
 }

 public void insert(Object data){
      rwl.writeLock().lock();
      try{
         // insert data into the database
      }finally{
         rwl.writeLock().unlock();
      }
 }
}
```

(b) Implemented with ReentrantReadWriteLock

```
class CachedData {
   ...
   synchronized void processCachedData() {
     if (!cacheValid) {
         //write to cache
     }
     //use data
   }
 }
```

(c) Implemented with synchronized

```
class CachedData {
  ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

  void processCachedData() {
        rwl.readLock().lock();
        if (!cacheValid) {
            rwl.readLock().unlock();
            rwl.writeLock().lock();
            try {
                if (!cacheValid) {
                    // write to cache
                    rwl.readLock().lock();
                }
            } finally { rwl.writeLock().unlock();}
        }
        try {
            // use data
        } finally {
            rwl.readLock().unlock();
        }
    } }
```

(d) Implemented with ReentrantReadWriteLock

Figure 1. (**a**) The method implemented with *synchronized* locks; (**b**) the method implemented with *ReentrantReadWriteLocks*; (**c**) the method implemented with *synchronized* locks; and (**d**) shows the method implemented with lock downgrading of *ReentrantReadWriteLocks*.

The method *CacheprocessData()* in Figure 1c implements the operation of the database. The data are used directly if the data exist in the cache. The data from the database are read and written to the local cache if the data do not exist in the local cache.

Figure 1d shows the method *processCachedData()* implemented through the lock downgrading mode of *ReentrantReadWriteLocks*. The code shows that the first read lock is acquired to read the data into the local cache. If the data do not exist in the local cache, the current thread will release a read lock to acquire a write lock and find the data in the database, then write the found data into the local cache. The thread finally acquires the read lock and then releases the write lock to complete lock downgrading.

3.3. Performance Evaluation

This section first compares the performance of the *synchronized* lock and the *ReentrantReadWriteLock*, then compares the results of the *synchronized* lock and lock downgrading of *ReentrantReadWriteLocks*.

Figure 2 shows the results of four code fragments executed under different configurations. We plot the execution time where *WT* represents the number of write threads and *RT* represents the number of read threads. All measuring results are obtained by calculating the mean value of 10 runs.

Figure 2a compares the results of operations using *ReentrantReadWriteLocks* and *synchronized* locks (the code in Figure 1a,b) with 10 total threads, each thread performing 100 operations). When $RT = 9$, and $WT = 1$ (WT represents the number of write threads and RT represents the number of read threads), *ReentrantReadWriteLocks* and *synchronized* locks have notable difference in execution time. However, the execution time is basically the same when $RT = 9$, and $WT = 1$. Figure 2b is the result of the execution of the code with a total of 100 threads, with 100 operations per thread performed, and the difference of the execution time between the *ReentrantReadWriteLock* and the *synchronized* lock is even more significant when $RT = 90$ and $WT = 10$.

Figure 2c presents the result of operations using *synchronized* locks and lock downgrading (the source code is similar to that in Figure 1c,d) with a total of 10 threads. Each thread executes operations 100 times. The read thread represents that the data exist in the local cache, so the data are used directly. The write thread represents that the data are not in the local cache and need to be written to the cache from the database. The figure shows that the impact of the number of read and write threads has not significant impact on the performance of using *synchronized* locks, but the execution time of the program using *ReentrantReadWriteLocks* decreases when the read threads increase. Figure 2d is the execution result under 100 threads, and the overall trend is similar to Figure 2c.

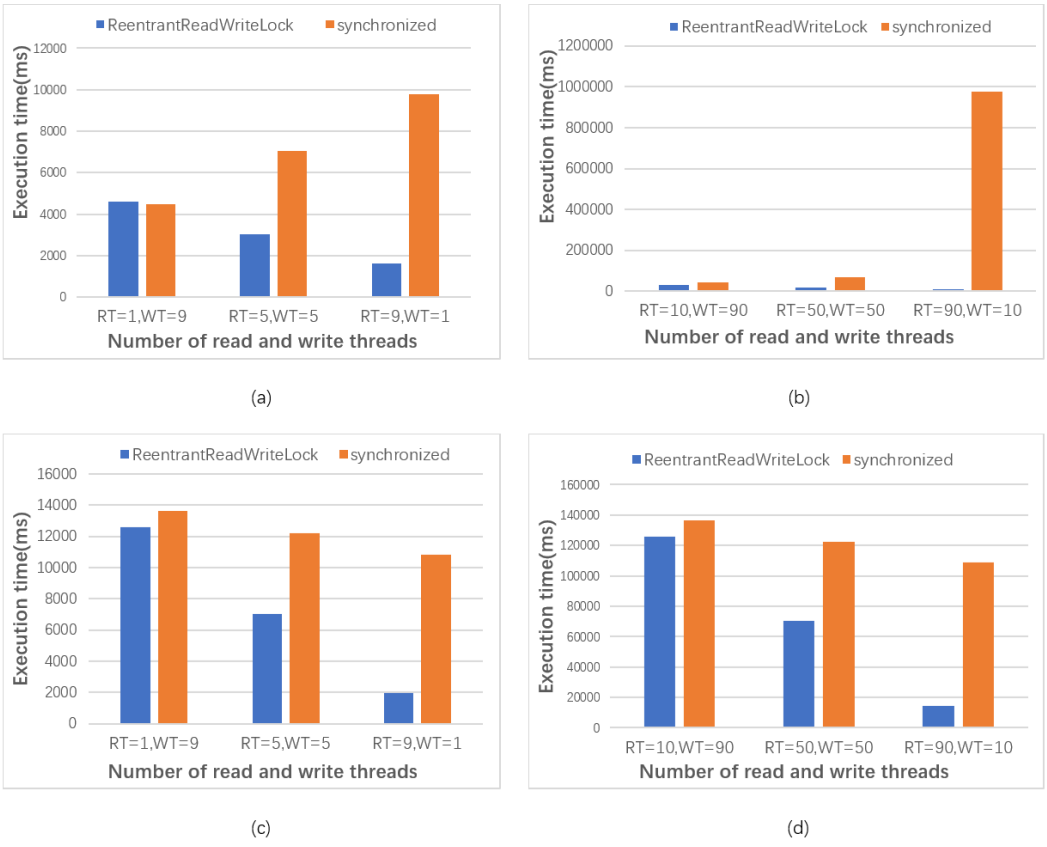

Figure 2. Performance results. (**a**) Results of synchronized lock and ReentrantReadWriteLock with 10 threads; (**b**) Results of synchronized lock and ReentrantReadWriteLock with 100 threads; (**c**) Results of synchronized lock and lock downgrading with 10 threads; (**d**) Results of synchronized lock and lock downgrading with 100 threads.

The results indicate that a program using *ReentrantReadWriteLocks* will perform better than that using *synchronized* locks when the read operations exceed the write operations and the read operation takes a relatively long time. When lock downgrading is employed, a program using *ReentrantReadWriteLocks* also shows better performance.

4. Refactoring for Fine-Grained ReentrantReadWriteLocks

4.1. Refactoring Framework

The refactoring framework is shown in Figure 3. WALA [32] was used to design our analysis algorithm. Visiting pattern analysis was employed to find the target code; alias analysis was used to check the alias of monitor objects; side-effect analysis was used to analyze the critical section and generate a character sequence. We designed five lock modes for fine-grained *ReentrantReadWriteLocks*, which could be inferred by our analysis and followed the inference rules.

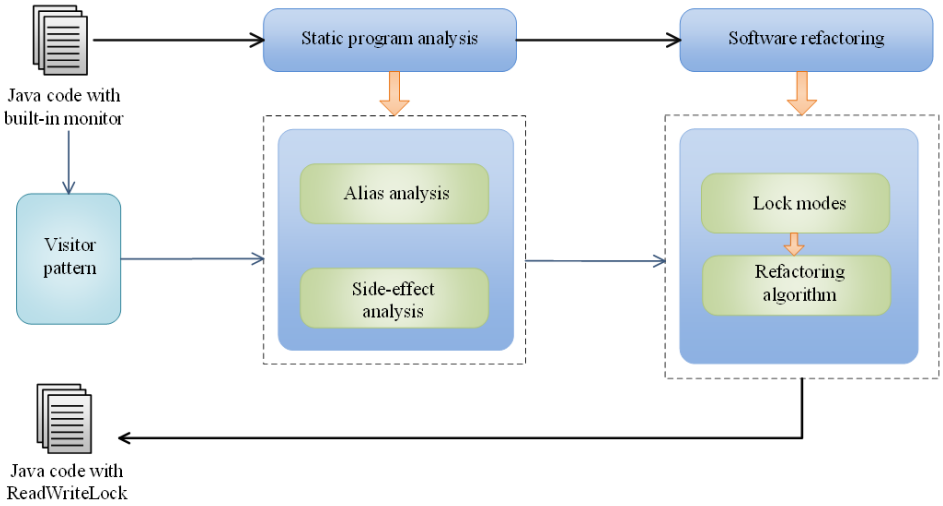

Figure 3. The refactoring framework.

4.2. Visitor Pattern Analysis

We parse the Java code into an abstract syntax tree (AST) through ASTParser [33] (a Java language parser for creating abstract syntax trees in Eclipse JDT). An AST node represents a Java source code construct, such as a name, type, expression, statement, or declaration. We use the visitor pattern to traverse all nodes on the AST and find all monitors in the program.

We must distinguish the built-in monitors and collect the monitor objects. For a synchronized lock, synchronizing methods and synchronized blocks should be considered separately, so should static methods and non-static ones.

For an object instance:

- For *synchronized* instance methods, the monitor object is *this*;
- For a *synchronized* block with an instance monitor object *o*, the monitor object is *o*;

For a class:

- For *synchronized* static methods, the monitor object is a class object;
- For a *synchronized* block with a static monitor object *O*, the monitor object is *O*;

For an object instance, we declare a new instance of *ReentrantReadWriteLock* in the class. For the monitor behavior that acts within the scope of a class, we declare a static *ReentrantReadWriteLock* instance in this class.

We define a *HashMap lockmap* to store the key-value pairs between the monitor object and the lock field, where the key is the monitor object, and the value is the corresponding lock field.

4.3. Lock Mode

Our refactoring tool transforms a built-in monitor into a fine-grained *ReentrantReadWriteLock* by lock downgrading and lock splitting.

Our refactoring tool directly applies the read locks to methods and synchronization blocks that have no side effects. For methods or blocks with side effects, the write locks, downgrading locks and splitting locks are used.

The fine-grained lock mode shown in Figure 4 is implemented by lock downgrading [29]. Figure 4a shows the program before refactoring. The method *cache()* will first judge the conditional variant *flag*. Only when the *flag* is true, the write operations will be executed.

Figure 4b presents the program after refactoring. Under the control of a read lock, a conditional statement *flag* is read (Line 4). If the condition is met, write operations are executed. Therefore, the current thread will release the read lock (Line 5) and acquire the write lock (Line 6) to perform the write operations. Note that a thread needs to release the read lock before getting the write lock. After acquiring the write lock, the conditional state will be rechecked (Line 7) in case other threads acquire the write lock and modify the state.

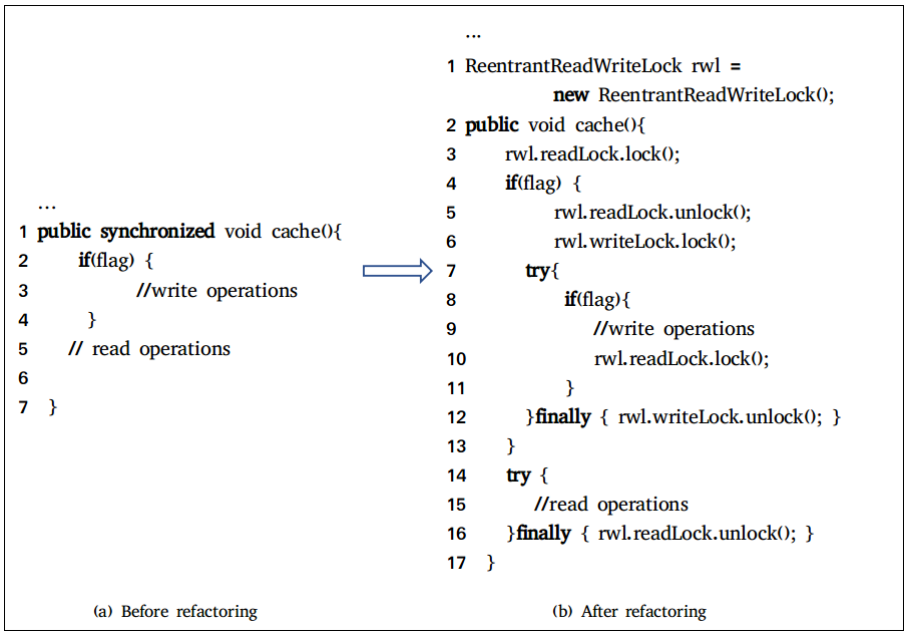

Figure 4. (**a**) Method *cache()* based on built-in monitors. (**b**) Method *cache()* using the lock downgrading of *ReentrantReadWriteLocks*

Figure 5 shows three fine-grained locking modes through splitting of the *ReentrantReadWriteLock*. Figure 5a shows the code before refactoring, and Figure 5b presents the code after refactoring. In the code of Figure 5b, the read lock is used to read the conditional statement (Line 5). If the conditional state is true, the read lock is released (Line 6), and then the write lock is acquired. The *finally* block will check what lock the thread is holding (Line 13). The write lock will be released when the write lock is held by the thread, and the read lock will be released when the read lock is held.

Other refactoring implementations shown in Figure 5c may cause threads to lose their perception of data updates. For instance, the section protected by the write lock has a write operation on the shared variable *s*, and the section protected by the read lock has a read operation on *s*. This will cause a problem: the thread may not read the data it has already modified. Synchronization problems may arise when the section protected by the write lock and the section protected by the read lock have the same shared variables.

We made a precondition that the read and write operations cannot access the same variable for refactoring under this mode. Because two operations access the same variable, the visibility of the data will be lost. For example, thread A acquires the read lock and reads the value of variable *i*, then releases the read lock. Thread B acquires write lock and modifies the value of variable *i*. But thread A cannot know the update of the value.

We put the shared variables read by the read lock into the list *readlist*, and the shared variables written by the write lock into the collection *writelist*. If the two lists do not share the same element, the refactoring tool uses the read lock. Otherwise, the tool uses the write lock for refactoring.

```
...
1 public synchronized void checkToWrite(){
2     if(flag){
3         //write operations
4     }
5 }
```

(a) Before refactoring

```
1 ReentrantReadWriteLock rwl =
            new ReentrantReadWriteLock();
2 public void checkToWrite(){
3     rwl.readLock.lock();
4     try {
5         if(flag) {
6             rwl.readLock.unlock();
7             rwl.writeLock.lock();
8              if(flag){
9                  // write operations
10             }
11        }
12    }finally {
13        if(rwl.isWriteLockedByCurrentThread()){
14            rwl.writeLock.unlock();
15        }else{
16            rwl.readLock.unlock();
17        }
18    }
19 }
```

(b) After refactoring

```
1 ReentrantReadWriteLock rwl =
                new ReentrantReadWriteLock();

2 public void WriteRead(){
3         rwl.writelock.lock();
4         try{
5             // write operations
6         }finally{
7         rwl.writelock.unlock();
8         }
9         rwl.readlock.lock();
10        try {
11            // read operations
12        }finally{rwl.readlock.unlock();}
13    }


14 public void ReadWrite (){
15        rwl.readlock.lock();
16        try {
17            //read operations
18        }finally{
19            readlock.unlock();
20        }
21        rwl.writelock.lock();
22        try{
23            //write operations
24        }finally{rwl.writelock.unlock();}
25    }
```

(c) Lock splitting

Figure 5. splitting of *ReentrantReadWriteLocks*

4.4. Alias Analysis

When synchronization blocks are transformed, our tool will analyze the lock set. The monitor objects of the synchronization blocks may have different names but two or more objects point to the same memory position. We use the program analysis framework WALA [32] to design our alias analysis to check alias on the lock set. Our alias analysis is based on context-sensitivity pointer analysis.

WALA uses a *HeapModel* to abstract pointers and heap locations and provides a *HeapGraph* to navigate the results of a pointer analysis. The nodes in a HeapGraph are *PointerKeys* and *InstanceKeys*. The *PointerKey* represents an abstract pointer and the *InstanceKey* represents an abstract heap location. There is an edge from a *PointerKey* to an *InstanceKey* when the*PointerKey* points to the *InstanceKey*, and there is an edge from an *InstanceKey* to a *PointerKey* when the *PointerKey* represents a field of an object instance modeled by the *InstanceKey*.

For example, we have a HeapGraph *h* and an *InstanceKey p* of a monitor object. We first use *h.getSuccNodes(p)* to find all pointer keys that *InstanceKeys p* may point to, and for each such *InstanceKey i*, *h.getPredNodes(i)* are used to find other *PointerKeys* that the alias *p* may point to. Our pointer analysis is based on this example.

4.5. Side Effect Analysis

An operation, method or expression has a side effect if it modifies the state outside its local environment. Our side effect analysis is to identify whether the critical section has side effects. WALA uses the Intermediate Representation (IR) structure to get all instructions in the method. The WALA IR is the central data structure that represents the instructions of a particular method. The IR represents a method's instructions in a language close to JVM bytecode, but in an SSA-based

register transfer language which eliminates the stack abstraction, it relies instead on a set of symbolic registers. As shown in the code in Figure 6, we analyze each instruction in the method and generate a sequence string for read and write operations for each method.

The side-effect analysis algorithm is shown in Figure 6. We first get all the instructions in the method and store them in a collection (Line 4), then traverse each instruction and analyze the side effects using the method *getAnalysis* (Line 14). The analysis method determines whether there is any instruction that modifies the memory. If the instruction is *InvokeInstruction*, the analysis method will get the called method, and the instruction in the method will be traversally analyzed. The method has side effects if it has a write instruction.

```
1 String sideEffectAnalysis(MethodReference m){
2     StringBuffer sb=new StringBuffer();
3     all instructions from m → instructions;
4     for(Instruction ins:instructions){
5         if(getAnalysis (ins)){
6             sb.append("W");
7         }else{
8             sb.append("R");
9         }
10    }
11    return sb.toString();
12 }
14 boolean getAnalysis (Instruction ins){
15    if(ins is a write instruction to static field||
          ins is a write instruction to instance field||
          ins is a write instruction to heap memory ){
16        return true;
17    }else if(ins is a InvokeInstruction){
18        get method ins_m of ins invoked
19        for each instruction ins_i from ins_m
20            if(getAnalysis (ins_i)) return true;
21        return false;
22    }
23    return false;
24 }
```

Figure 6. Algorithm for side-effect analysis

As *ReentrantReadWriteLocks* have many types of locking modes, such as read/write locks, lock degrading and lock splitting, the use of lock modes depends on the side effects of the critical section. The side-effect analysis will analyze the critical section and generate a character sequence.

To match the character sequence, we define five regular expressions for inferring lock modes. The regulation sequence and representation of the characters are shown in Table 1.

Table 1. Regulation sequence and character representation.

Regulation	Sequence
Regulation 1	R^+
Regulation 2	$((C\|T)^*\|(R\|W)^*)^*$
Regulation 3	$R^*CR^*W(W\|R)^*T$
Regulation 4	$R^+W^+\|W^+R^+$
Regulation 5	$R^*CR^*W(W\|R)^*TR^+$

R: Read operation; W: Write operation; C: If condition; T: End of if condition; *: Zero or multiple times; $^+$: Once or multiple times.

Regulation 1: The read lock mode.

Regulation 1 shows the read lock mode, in which the critical section has at least one read operation and does not have write operations.

Regulation 2: The write lock mode.

Regulation 2 represents a mode in which the critical section has at least one write operation.

Regulation 3: The lock downgrading mode.

Regulation 3 represents a mode in which a critical section has an *if* statement that has write operations and at least one read operation in the end.

Regulation 4: The lock splitting mode.

Regulation 4 represents a mode in which a critical section only has one *if* statement and has write operations in the body of statement.

Regulation 5: The lock splitting mode.

Regulation 5 represents the separation of read and write operations in the critical section.

We now describe the refactoring algorithm in more detail. The code in Figure 7 first gets the monitor object of the method *m* (Line 2), and then checks it in the lock set *lockmap*. If the monitor object exists in *lockmap*, the corresponding lock field is obtained. Otherwise, it creates an appropriate lock field based on the type of the monitor object and presents a new mapping relationship in *lockmap*.

After the lock field is obtained, the synchronizing methods and synchronized blocks are refactored accordingly. For a synchronizing method (Line 19), the method is analyzed first by side effect analysis, and the corresponding reading and writing sequence is generated. A finite automaton is used to identify the reading and writing sequence (Line 20). Because the monitor object in the synchronized block may have aliases, alias analysis is included in all methods for refactoring the synchronized blocks (Line 27).

```
1 refactoringtoRWlock(MethodReference m){
2     get the monitor expression ex from m;
3     Field lock;
4     if(lockmap.cotainKey(ex)){
5             lock ← lockmap.get(ex);
6     }else{
7         if ex is an instance object
8             lock ← create an instance field;
9             lockmap.put(ex,lock);
10        else
11            lock ← create a static field;
12            lockmap.put(ex,lock);
13    }
14    if m is a synchronized method
15        refactoringtoMethod(m.lock);
16    else if m has synchronized blocks
17        refactoringtoBlock(m,lock);
18 }

19 refactoringtoMethod(MethodReference m,Field lock){
20     Sting str=getEffectAnalysis(m);
21     remove synchronized modifier from m;
22     use Finite Automata to recognize the str;
23         Refactoring according to the regulation;
24 }

25 refactoringtoBlock(MethodReference m,Field lock){
26     Sting str= getEffectAnalysis(m);
27     may_alias_analysis();
28     use Finite Automata to recognize the str;
29         Refactoring according to the regulation;
30 }
```

Figure 7. Refactoring algorithm for fine-grained *ReentrantReadWriteLocks*.

5. Refactoring Tool and Practical Issues

5.1. Refactoring Tool

We implement our refactoring tool as an Eclipse plugin (the source code and the jar of the repository are available at https://uzhangyang.github.io/refactoring.html). Figure 8 is a screenshot of our refactoring tool, which displays a comparison between the code before and after refactoring. The source code before refactoring is presented on the left while that after refactoring is on the right.

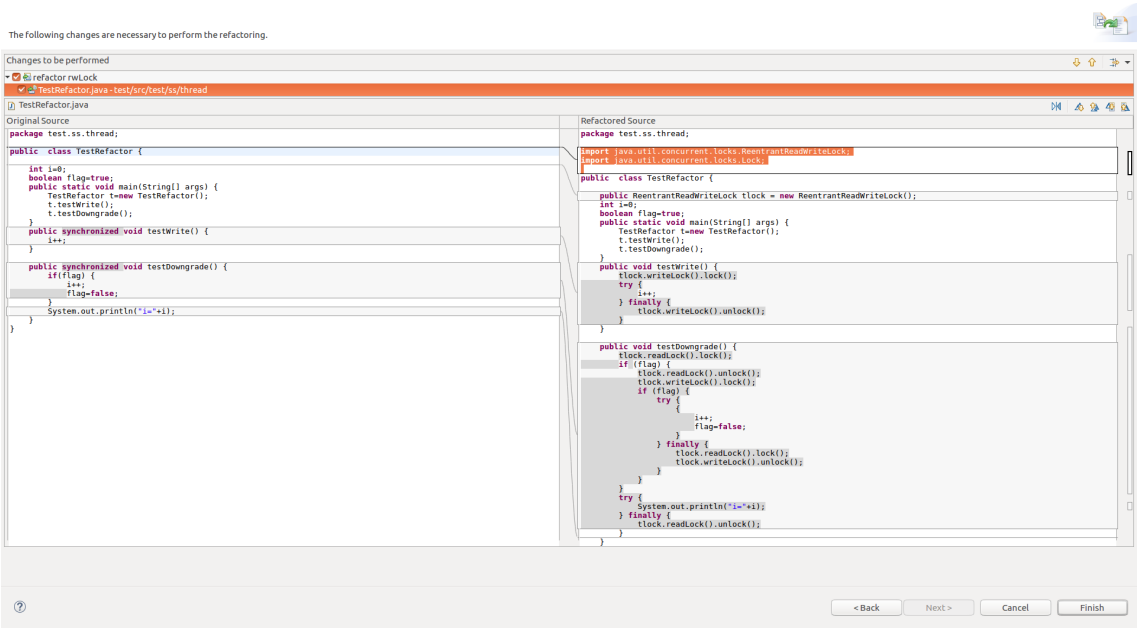

Figure 8. Screenshot of the refactoring tool.

5.2. Practical Issues

As our refactoring tool inserts part of the code into the try-finally construct, the scope of variables may be changed for some variables defined in the critical section. To resolve this problem, our tool checks these defined variables and allows them to be defined outside of the try block.

6. Evaluation

6.1. Setup

All experiments are conducted on a laptop with a 1.6 GHz Intel Core i5 CPU, 8 GB RAM. The machine runs Ubuntu 16.04 and has JDK 1.8.0_191, Eclipse 4.10.0 and WALA [32] 1.5.2 installed.

6.2. Benchmarks

Several real-world applications, including HyperSQL DataBase [34](HSQLDB), Cassandra [35], JGroups [36], Freedomotic [37] and Multipurpose Infrastructure for Network Applications(MINA) [38], are selected to evaluate our refactoring tool.

HSQLDB, Cassandra, and JGroups are widely used, and contain a lot of built-in monitors. HSQLDB is fully multi-threaded and supports high performance 2PL and MVCC (multiversion concurrency control) transaction control models. HSQLDB is used as a database and persistence engine in over 1700 Open Source Software projects and many commercial products. Cassandra is an Apache distributed database. It can be used to manage large amounts of structured data. Cassandra is the most commonly used NoSQL database. Because the data provided by the IoT are time series, Cassandra is often used to store data generated by sensors and devices in IoT applications. Over 1500 more companies worldwide with massive, active data sets are using Cassandra. JGroups is a reliable group communication tool written by Java. It is widely used in distributed systems, including JBoss, ElasticMQ, etc.

Freedomotic is an open-source, flexible, and secure IoT application framework for building and managing modern smart spaces. Freedomotic can run on Raspberry Pi and can easily interact with DIY Arduino projects. It is widely used in IoT applications. Apache MINA is a network application framework that helps developers develop high-performance, high-scalability network applications. MINA comes with many sub-projects such as AsyncWeb, FtpServer, SSHD, etc.

6.3. Results

Table 2 presents the evaluation results, with parameters including the lines of source code and built-in monitors in each benchmark. The last four columns show the number of four lock modes of each benchmark refactored by our tool.

Table 2. Evaluation of the refactoring.

Benchmark	KSLOC	#Built-In Monitors	#Downgrading Locks	#Splitting Locks	#Read Locks	#Write Locks
HSQLDB	179	621	6	39	51	525
Cassandra	432	239	2	24	33	180
JGroups	123	179	5	33	28	113
Freedomotic	57	21	2	1	2	16
MINA	24	12	0	3	1	8

Our refactoring tool detected and refactored 621 built-in monitors in HSQLDB, 239 in Cassandra, 179 in JGroups, 21 in Freedomotic, and 21 in MINA. In total, all benchmarks have 1072 built-in monitors refactored by our refactoring tool. 12,008 SLOCs were modified. These results show that our refactoring tool can effectively save the developers' time and energy.

For IoT applications, our tool has refactored Freedomotic—an IoT framework—and Cassandra, a widely used database engine. Our tool is not running in the IoT environment, but our tool can refactor IoT concurrent software, which runs in the IoT environment. We conclude that there are programs in our real-world applications that conform to our lock downgrading and lock splitting rules. In most cases, the tool uses write locks, and there is not much lock downgrading. We don't suggest converting all built-in monitors into *ReentrantReadWriteLocks*, because the performance of the *ReentrantReadWriteLock* is not necessarily better than the built-in monitor. The actual situation should be considered during refactoring.

6.4. Correctness Of Refactoring

HSQLDB benchmark is evaluated by connecting the database under several connection modes, such as in-memory mode, standalone mode and server mode. The evaluation results show that HSQLDB under all modes can connect to the database. We also create database, run SQL statements to insert and delete data, and perform other database operations. They all execute correctly. We run the *JDBCBench* and *TestBench* in package *org.hsqldb.test*. The *JDBCBench* is a test of JDBC connection and *TestBench* is a stress test of transaction processing. They all run correctly and return a benchmark report.

For Cassandra, we connect to the database and executed some CQL statements after the refactoring. They all work correctly, and we run all the unit tests in the test folder of Cassandra. A total of 648 unit tests, cover almost all classes in the source code. We find that they all run smoothly without reporting any errors. Cassandra has part of code that already use *ReentrantReadWriteLock*. We then manually refactor them back to synchronized locks, and use our tool to infer the original *ReentrantReadWriteLock* usage. The method *mayReload()* in class *CompactStrategyManager* uses write lock before refactoring. Our tool infers this method uses splitting lock. After manual check, the splitting lock does not change the behavior of method *mayReload()*. The other locks are inferred as same as original usage.

We used JGroups to deploy the cluster of three nodes after refactoring and successfully completed the communication between them. There are some test programs in JGroups, 49 of which were tested, and they all ran smoothly without reporting any errors.

Because Freedomotic and MINA have fewer built-in monitors, we manually inspected all the refactored locks. We manually identified (1) if the refactoring had changed the behavior; (2) if a correct kind of lock was inferred, (3) if a lock was inserted to a correct position, (4) if a lock structure was used correctly, and (5) if the critical section was protected safely. During the inspection, we found that

the refactoring had not changed any behavior of the original programs, and each critical section had been inferred with the kind of lock according to the lock mode and almost all of them were accurate. The position that the lock inserted and the used lock structure were correct. Finally, the critical sections are surrounded by locks and the protection of the critical section is safe.

6.5. Comparison With Relocker

Max et al. [14] have proposed an algorithm of refactoring for *ReentrantLocks* and *ReentrantReadWriteLocks*, as well as a refactoring tool Relocker.

Running *Relocker* requires an earlier JDK version, so the JDK version used in this experiment is 1.6. HSQLDB version is 1.8.0.10, and Cassandra version is 0.4.0. Table 3 shows the comparison result between Relocker and our tool. Compared with Relocker which only uses read locks or write locks for synchronization protection, our tool uses lock downgrading and lock splitting to realize more fine-grained locking. Our tool has inferred more read locks than Relocker. After manual verification, the read locks inferred by our tool is used correctly. Relocker still relies heavily on manual selection of codes to refactor, and our tool is more automatic.

Table 3. Comparison between Relocker and our tool.

Benchmark	Relocker			Our Tool			
	#Read Locks	#Write Locks	#Can't be Refactoring	#Downgrading Locks	#Splitting Locks	#Read Locks	#Write Locks
HSQLDB	31	212	23	8	23	45	190
Cassandra	4	50	3	3	1	6	47

7. Conclusions and Future Work

The JDK library provides flexible locking constructs that can improve performance of software by reducing lock contention. In this paper, we presented an approach might improve the software quality by using *ReentrantReadWriteLocks*. We proposed a refactoring algorithm for fine-grained *ReentrantReadWriteLocks*, and implemented a refactoring tool as an Eclipse plugin. Our tool has been tested by several real-world applications. The refactoring approach is applicable not only to IoT software, but also to other concurrent software.

The major limitation of this study is that the selected applications cannot represent all applications which may have different concurrent behaviors. In future studies, we will use our refactoring tool to refactor more IoT programs, find more application scenarios suitable for fine-grained read–write locking, and explore more refactoring modes that reduce lock contention.

Author Contributions: Designed the framework of detection method and paper writing, Y.Z.; Wrote the first draft, coding, and experiment, S.S.; Setting overall research goals and part of the ideas, M.J.; Literature search, data analysis, and proof read, J.Q. and Z.T.; Verify the model, and proof read, X.D. and M.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research is supported by the Guangdong Province Key Research and Development Plan (2019B010137004), the National Key research and Development Plan (2018YEB1004003), the National Natural Science Foundation of China (U1636215,61871140,61872100), in part by the Scientific Research Foundation of Hebei Educational Department under Grant ZD2019093, in part by the Fundamental Research Foundation of Hebei Province under Grant 18960106D, and Guangdong Province Universities and Colleges Pearl River Scholar Funded Scheme (2019).

Acknowledgments: The authors gratefully acknowledge the helpful comments and suggestions from the reviewers.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Shen, M.; Ma, B.; Zhu, L.; Mijumbi, R.; Du, X.; Hu, J. Cloud-based approximate constrained shortest distance queries over encrypted graphs with privacy protection. *IEEE Trans. Infor. Forensics Secur.* **2017**, *13*, 940–953. [CrossRef]
2. Xiao, L.; Li, Y.; Huang, X.; Du, X. Cloud-based Malware Detection Game for Mobile Devices with Offloading. *IEEE Trans. Mob. Comput.* **2017**, *16*, 2742–2750. [CrossRef]
3. Hassan, W. H. Current research on Internet of Things (IoT) security: A survey. *Comput. Netw.* **2019**, *148*, 283–294.
4. Tian, Z.; Luo, C.; Qiu, J.; Du, X.; Guizani, M. A distributed deep learning system for Web attack detection on edge devices. *IEEE Trans. Ind. Inform.* **2019**. [CrossRef]
5. Tian, Z.; Su, S.; Shi, W.; Du, X.; Guizani, M.; Yu, X. A data-driven method for future Internet route decision modeling. *Future Gener. Comput. Syst.* **2019**, *95*, 212–220. [CrossRef]
6. Qiu, J.; Chai, Y.; Tian, Z.; Du, X.; Guizani, M. Automatic Concept Extraction based on Semantic Graphs from Big Data in Smart City. *IEEE Trans. Comput. Soc. Syst.* **2019**. [CrossRef]
7. Qiu, J.; Du, L.; Zhang, D.; Su, S.; Tian, Z. Nei-TTE: Intelligent Traffic Time Estimation Based on Fine-grained Time Derivation of Road Segments for Smart City. *IEEE Trans. Ind. Inform.* **2019**. [CrossRef]
8. Tan, Q.; Gao, Y.; Shi, J.; Wang, X.; Fang, B.; Tian, Z. Toward a Comprehensive Insight Into the Eclipse Attacks of Tor Hidden Services. *IEEE Internet Things J.* **2019**, *6*, 1584–1593. [CrossRef]
9. Aldrich, J.; Chambers, C.; Sirer, E.; Eggers, S. Static analyses for eliminating unnecessary synchronization from Java programs. *Int. Static Anal. Symp.* **1999**, *1694*, 19–38.
10. Bogda, J.; Hölzle, U. Removing unnecessary synchronization in Java. *ACM Sigplan Not.* **1999**, *34*, 35–46. [CrossRef]
11. Tao, B.; Qian, J. Refactoring java concurrent programs based on synchronization requirement analysis. *IEEE Int. Conf. Softw. Maint. Evol.* **2014**, 361–370.
12. Lea, D. *Concurrent Programming in Java: Design Principles and Patterns*; Addison-Wesley Professional: Boston, MA, USA, 2000.
13. Diniz, P. C.; Rinard, M. C. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel Distrib. Comput.* **1998**, *49*, 218–244. [CrossRef]
14. Schafer, M.; Sridharan, M.; Dolby, J.; Tip, F. Refactoring Java programs for flexible locking. In Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; pp. 71–80.
15. Zhang, Y.; Dong, S.; Zhang, X.; Liu, H.; Zhang, D. Automated Refactoring for StampedLock. *IEEE Access* **2019**, *7*, 104900–104911. [CrossRef]
16. Bavarsad, A.G.; Atoofian, E. Read-Write Lock Allocation in Software Transactional Memory. In Proceedings of the 2013 42nd International Conference on Parallel Processing, Lyon, France, 1–4 October 2013; pp. 680–687.
17. Emmi, M.; Fischer, J. S.; Jhala, R.; Majumdar, R. Lock allocation. *ACM Sigplan Not.* **2007**, *42*, 291–296. [CrossRef]
18. Kawachiya, K.; Koseki, A.; Onodera, T. Lock reservation: Java locks can mostly do without atomic operations. *ACM Sigplan Not.* **2002**, *37*, 130–141. [CrossRef]
19. Hofer, P.; Gnedt, D.; Schörgenhumer, A.; Mössenböck, H. Efficient tracing and versatile analysis of lock contention in Java applications on the virtual machine level. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, Delft, The Netherlands, 12–16 March 2016; pp. 263–274.
20. Zhang, Y.; Shao, S.; Liu, H.; Qiu, J.; Zhang, D.; Zhang, G. Refactoring Java Programs for Customizable Locks Based on Bytecode Transformation. *IEEE Access* **2019**, *7*, 66292–66303. [CrossRef]
21. Dig, D.; Marrero, J.; Ernst, M.D. Refactoring sequential Java code for concurrency via concurrent libraries. In Proceedings of the 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16–24 May 2009; pp. 397–407.
22. Wloka, J.; Sridharan, M.; Tip, F. Refactoring for reentrancy. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering, Amsterdam, The Netherlands, 24–28 August 2009; pp. 173–182.

23. Du, X.; Guizani, M.; Xiao, Y.; Chen, H.H. Transactions papers a routing-driven elliptic curve cryptography based key management scheme for heterogeneous sensor networks. *IEEE Trans. Wirel. Commun.* **2009**, *8*, 1223–1229. [CrossRef]
24. Dong, P.; Du, X.; Zhang, H.; Xu, T. A detection method for a novel DDoS attack against SDN controllers by vast new low-traffic flows. In Proceedings of the 2016 IEEE International Conference on Communications (ICC), Kuala Lumpur, Malaysia, 22–27 May 2016; pp. 1–6.
25. Tian, Z.; Shi, W.; Wang, Y.; Zhu, C.; Du, X.; Su, S.; Sun, Y.; Guizani, M. Real-Time Lateral Movement Detection Based on Evidence Reasoning Network for Edge Computing Environment. *IEEE Trans. Ind. Inform.* **2019**, *15*, 4285–4294. [CrossRef]
26. Qiu, J.; Chai, Y.; Tian, Z.; Du, X.; Guizani, M. Vcash: A Novel Reputation Framework for Identifying Denial of Traffic Service in Internet of Connected Vehicles. *IEEE Internet Things J.* **2020**. [CrossRef]
27. Xiao, L.; Wan, X.; Dai, C.; Du, X.; Chen, X.; Guizani, M. Security in mobile edge caching with reinforcement learning. *IEEE Wirel. Commun.* **2018**, *25*, 116–122. [CrossRef]
28. Wu, L.; Du, X.; Wang, W.; Lin, B. An Out-of-band Authentication Scheme for Internet of Things Using Blockchain Technology. In Proceedings of the 2018 International Conference on Computing, Networking and Communications (ICNC), Maui, HI, USA, 5–8 March 2018; pp. 769–773.
29. Oracle. Java.util.concurrent.locks.ReentrantReadWriteLock API Specification. Available online: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html (accessed on 5 December 2019).
30. Oracle. Java.util.concurrent.locks API Specification. Available online: http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/package-summary.html (accessed on 12 August 2019).
31. Pinto, G.; Torres, W.; Fernandes, B.; Castor, F.; Barros, R.S. A large-scale study on the usage of Java's concurrent programming constructs. *J. Syst. Softw.* **2015**, *106*, 59–81. [CrossRef]
32. WALA. Available online: http://wala.sourceforge.net/wiki/index.php/Main_Page (accessed on 12 August 2019).
33. Eclipse JDT. Org.eclipse.jdt.core.dom.ASTParser API Specification. Available online: https://help.eclipse.org/kepler/ntopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html (accessed on 6 December 2019).
34. HyperSQL. Available online: http://hsqldb.org/ (accessed on 6 December 2019).
35. Cassandra. Available online: https://cassandra.apache.org/ (accessed on 6 December 2019).
36. JGroups. Available online: http://www.jgroups.org/ (accessed on 6 December 2019).
37. Freedomotic. Available online: https://freedomotic-user-manual.readthedocs.io/en/latest/index.html (accessed on 6 December 2019).
38. MINA. Available online: http://mina.apache.org/mina-project/ (accessed on 6 December 2019).