FineLock: Automatically Refactoring Coarse-Grained Locks into Fine-Grained Locks

Yang Zhang
School of Information Science and Engineering,
Hebei University of Science and Technology
Shijiazhuang, China
zhangyang@hebust.edu.cn

Shuai Shao
School of Information Science and Engineering,
Hebei University of Science and Technology
Shijiazhuang, China
shao724854691@163.com

Juan Zhai
Rutgers University
Piscataway, USA
juan.zhai@rutgers.edu

Shiqing Ma
Rutgers University
Piscataway, USA
shiqing.ma@rutgers.edu

ABSTRACT

Lock is a frequently-used synchronization mechanism to enforce exclusive access to a shared resource. However, lock-based concurrent programs are susceptible to lock contention, which leads to low performance and poor scalability. Furthermore, inappropriate granularity of a lock makes lock contention even worse. Compared to coarse-grained lock, fine-grained lock can mitigate lock contention but difficult to use. Converting coarse-grained lock into fine-grained lock manually is not only error-prone and tedious, but also requires a lot of expertise. In this paper, we propose to leverage program analysis techniques and pushdown automaton to automatically covert coarse-grained locks into fine-grained locks to reduce lock contention. We developed a prototype *FineLock* and evaluates it on 5 projects. The evaluation results demonstrate *FineLock* can refactor 1,546 locks in an average of 27.6 seconds, including converting 129 coarse-grained locks into fine-grained locks and 1,417 coarse-grained locks into read/write locks. By automatically providing potential refactoring recommendations, our tool saves a lot of efforts for developers.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Refactoring, Fine-grained lock, Static analysis, Read-write lock, Pushdown automaton

ACM Reference Format:
Yang Zhang, Shuai Shao, Juan Zhai, and Shiqing Ma. 2020. FineLock: Automatically Refactoring Coarse-Grained Locks into Fine-Grained Locks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3395363.3404368

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3404368

1 INTRODUCTION

Lock is frequently used to guarantee the correctness of shared resources access in concurrent programs. To enforce exclusive access to a shared resource, only one thread can acquire the lock of a given resource at a time while other threads have to wait for the thread to release the lock. Unfortunately, concurrent programs based on locks are susceptible to lock contention which would result in low performance and poor scalability. To avoid lock contention, software transactional memory and lock-free algorithm have been proposed to control access to shared memory in concurrent programs. However, lock is still one of the most popular synchronization mechanisms among developers, which motivates us to provide automated support in refactoring lock-related code to reduce lock contention.

Inappropriate granularity of a lock makes lock contention even worse. It is generally accepted that employing a coarse-grained lock may exacerbate lock contention while a fine-grained lock may mitigate lock contention by decreasing the waiting time of other threads that attempt to acquire the lock. Consequently, there is a strong need to employ fine-grained lock. However, writing a program based on fine-grained lock is much more difficult compared with that based on a coarse-grained lock, which requires careful design and expertise. For example, developers have to consider which code patterns are applicable for fine-grained locks. Furthermore, when developers are required to maintain existing code that have coarse-grained locks, it would be error-prone and tedious to convert existing coarse-grained locks to fine-grained ones manually. This inspires us to develop an automation tool to help developers refactor code by converting coarse-grained locks into find-grained lock with the hope of reducing lock contention.

Some existing tools have been proposed to refactor locks automatically. McCloskey et al.[8] presented an automated tool *Autolocker* to convert pessimistic atomic sections into standard lock-based code. Schäfer et al. [10] presented a refactoring tool *Relocker* to convert a synchronized lock into a *ReentrantLock* and a *ReentrantReadWriteLock*. Both *Autolocker* and *Relocker* conducted refactoring for coarse-grained locks. Moreover, *Relocker* takes the critical section as a whole to analyze the read/write operations. Without performing detailed analysis for every operation in a critical section, *Relocker* is incapable of converting locks into fine-grained locks, which is confirmed by experiments on several benchmarks.

Zhang et al. [12] presented an automated refactoring tool *CLOCK* to convert a synchronized lock into a *StampedLock*. *CLOCK* could not only transform read and write locks, but also refactor downgrading/upgrading and optimistic read locks. However, *CLOCK* could conduct limited refactoring due to the non-reentrency of *StampedLock*. Some commercial refactoring tools, such as concurrency-oriented refactoring for JDT [9] and LockSmith [11], had been integrated into Eclipse and IntelliJ to merge locks, convert them, and make the field atomic. However, none of the existing tools is able to downgrade a coarse-grained synchronized lock into a fine-grained read-write lock.

Automatically refactoring a coarse-grained lock into a fine-grained read-write lock faces a few challenges. Firstly, automatically inferring an appropriate fined-grained lock for a critical section is extremely difficult, even for humans, which requires expertise. For example, a downgrading lock firstly uses a write lock and then employs a read lock. This entails us to automatically identify read operations and write operations from source code. Secondly, automatically refactoring into fine-grained lock requires to modify existing code without introducing new bugs. For example, it needs to locate a right position to create a lock object and ensure the lock is used correctly.

In this paper, we propose a novel solution for converting a coarse-grained synchronized lock into a fine-grained *ReentrantReadWriteLock* lock and develop a prototype named *FineLock*. Firstly, we leverage program analysis techniques to identify critical sections that can be refactored. Then we automatically identify read statements and write statements in the critical sections, and summarize them as a pattern (a sequence of characters where each character indicates whether a statement is a *read* operation or a *write* operation). After that, we feed the pattern into a pushdown automaton to decide what kind of refactor should be performed. Finally, based on the result from the automaton, a coarse-grained lock is transformed into a fine-grained lock including a downgrading lock and a splitting lock. We develop a prototype *FineLock* as an Eclipse plugin. We evaluate it on 5 real-world projects. The results show *FineLock* refactors 1,546 locks in an average of 27.6 seconds, including converting 129 coarse-grained locks into fine-grained locks and 1,417 coarse-grained locks into read/write locks, which improves throughput for these projects.

2 MOTIVATION

We showcase how our approach can reduce lock contention using the examples in Figure 1. Figure 1(a) shows a typical implementation of cache processing where the method is protected by a synchronized lock (line 1). It first checks whether the cache contains data in line 2. If so, the data is read (line 5). Otherwise, data will be written into the cache (line 3). Figure 1(b) presents the code refactored by *Relocker* [10]. According to the lock inference strategy of *Relocker*, a coarse-grained write lock is inferred (lines 22 and 29). However, the write operation is executed only when the cache does not have the data. The refactored code by *Relocker* is still too coarse-grained and has low concurrency. To allow more concurrency and reduce lock contention, we propose our approach to infer fine-grained locks. For the code in Figure 1(a), a read lock is inferred using our approach and the result code is shown in Figure 1(c). It first acquires

```
1  public synchronized void cached() {
2      if (!cacheValid) {
3          ··· // write into cache
4      }
5      ··· // read data
6  }
```
(a)

```
21 public void cached() {
22     lock.writeLock().lock();
23     try {
24         if (!cacheValid) {
25             ··· // write into cache
26         }
27         ··· // read data
28     } finally {
29         lock.writeLock().unlock();
30     }
31 }
```
(b)

```
41 public void cached() {
42     lock.readLock().lock();
43     if (!cacheValid) {
44         lock.readLock().unlock();
45         lock.writeLock().lock();
46         try {
47             if (!cacheValid) {
48                 ··· // write into cache
49                 lock.readLock().lock();
50             }
51         } finally {
52             lock.writeLock().unlock();
53         }
54     }
55     try {
56         ··· // read data
57     } finally {
58         lock.readLock().unlock();
59     }
60 }
```
(c)

Figure 1: The motivating example

a read lock (line 42) which is enough to ensure concurrency for the condition checking in line 43. When the condition holds, a written lock must be used and thus the read lock is released (line 44) and a write lock is acquired (line 45). Note that the condition is rechecked in line 47 to guarantee the consistency since the condition might be changed by another thread. After the data is written to the cache (line 48), the write lock is downgraded into a read lock (line 52) to allow more concurrency. Finally, after the data is read (line 56), the read lock is released in line 58. The example demonstrates that our tool can reduce lock contention compared to existing tools by converting coarse-grained locks into fine-grained locks.

3 DESIGN

Figure 2 gives the design of our approach. *FineLock* takes the source code based on coarse-grained locks as an input. It firstly analyzes each critical section by visitor pattern analysis, and then collects synchronized methods and blocks, as well as monitor objects. *FineLock* judges whether monitor objects are aliased or not. Based on results of alias analysis, it can generate a *ReentrantReadWriteLock* in each class. Secondly, read/write sequence of each critical section is obtained by side effect analysis. And then a pushdown automaton is defined to identify and to accept these read-write sequence. Both read/write lock and fined-grained lock including downgrading lock and splitting lock are recommended for each critical section. Finally, based on these analysis results, *FineLock* convert each synchronized method and block into fine-grained *ReentrantReadWriteLock*.

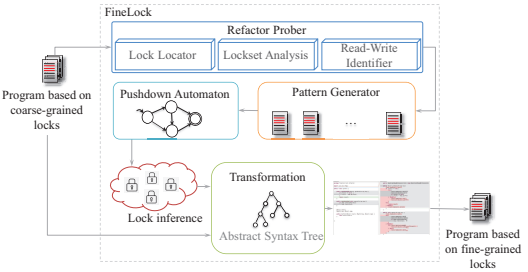

Figure 2: Overview of *FineLock*

3.1 Refactor Prober

Refactor prober is to locate synchronized locks that can be refactored into fine-grained locks and summarize the statements in a

critical section into a pattern which will be used by pushdown automaton to infer locks. Here a pattern means a sequence of characters where each character indicates whether a statement is a read operation or a write operation.

Lock Locator. Given a project, we first generate an abstract syntax tree (AST) using Eclipse JDT [4] for each source file and then leverage visitor pattern to locate all synchronized methods/blocks with their monitor objects by traversing AST.

Lockset Analysis. *FineLock* constructs a hash set *LockSet* in which a monitor object is the key and a *ReentrantReadWriteLock* object is the value. We leverage Wala [6] to perform alias analysis for each monitor object since the same monitor object in the original program should use the same *ReentrantReadWriteLock* instance in the refactored program.

Read-Write Identifier. For each statement in a critical section, the read-write identifier would identify it as a read operation or a write operation by conducting side-effect analysis. If a write lock is used to synchronized a read operation, it would downgrade the performance. This entails us to use an appropriate lock that is enough to ensure the correctness of concurrent programs, which means a statement should be identified as read or write. In *FineLock*, the following statements are identified as write operations: 1) a statement that modifies a static field or variant, 2) a method invocation whose callee contains a write operation, and 3) a library method whose code is unavailable or cannot be determined.

Pattern Generator. The pattern generator is to encode the sequence of programming statements into a sequence of characters where each character marks the state of the statement. Each character sequence is called a pattern and it would be feed into automaton to infer an appropriate lock. An identified read operation is marked as *r* and an identified write statement is marked as *w*. Also, *FineLock* marks *if* statements since we need to know the read/write property of the condition checking expression in the *if* statement to infer locks. The beginning of an *if* statement is marked as *c* and the end of an *if* statement is marked as *e*.

3.2 Pushdown Automaton

The pushdown automaton is utilized to infer locks based on a generated pattern for a critical section. Each character in the pattern sequence is used as a trigger to transfer from one state to another. The state transition diagram is presented in Figure 3.

The pushdown automaton $M_{fg} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a seven-tuple. $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ is a finite set of states and q_0 is the start state. $\Sigma = \{r, w, c, e\}$ is the input alphabet (defined in Section 3.1). $\Gamma = \{Z_0, V, C, D, A, B\}$ is the stack alphabet, and each value indicates the current status of the inferring process. Z_0 indicates an empty stack. V indicates halting the automaton, which means the given pattern does not meet any refactoring condition, and thus a write lock is inferred for the sake of safety. C, D, A, and B separately indicate the status based on the traversed sequence, where C indicates the traversed sequence is in an *if* statement, D indicates the traversed sequence contains a downgrading lock. A and B indicate splitting locks. Specifically, A means a read lock followed by a write lock and B means a write lock followed by a read lock. The state transition δ is a mapping set $Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^+$ where Γ^+ is the positive closure of Γ. δ is defined as $< q, x, X, q', T >$ where q is a state, x is an element of Σ, X is a stack symbol and T is a stack operation. It means if the current state is q, the input is x and the top element of the stack is X, we would transfer to state q', and execute stack operation T. The stack operation T consists of X, $X'X$, and ρ, where X indicates transition without stack operation, $X'X$ indicates pushing X' into the stack, and ρ indicates popping the top element of the stack. To simplify the representation, we use $< x, X/T >$ to represent for the transition $< q, x, X, q', T >$. F is a finite set of final states and $F = \{q_1, q_2, q_3, q_4, q_5, q_6\}$, where q_1 and q_2 indicates that a read lock and a write lock is inferred respectively, q_3, q_4 and q_5 indicate a splitting lock is inferred, and q_6 indicates a downgrading lock is inferred.

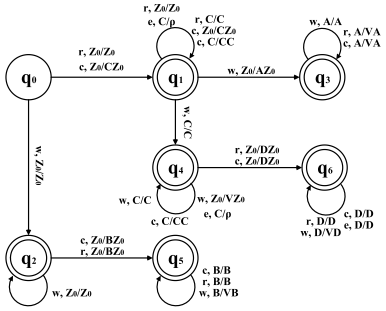

Figure 3: Pushdown Automaton

Examples. Take the code in Figure 1(a) as an example. The pattern sequence "*cwer*" is generated for the method *cached()* in line 1 and it is used as the input for pushdown automaton to infer refactored locks. To start with, the stack is initialized as empty using the symbol Z_0. The first character of the given pattern is *c*, which indicates a condition evaluation of an *if* statement, and thus we transfer to state q_1 and push symbol C into the stack. Given the next character *w* and the top symbol C, it implies that the write operation is in an *if* statement, and we can transfer from state q_1 to state q_4. In the next pass, we have *e* as the input and the top symbol of the stack is C, which indicates the end of the *if* statement. Hence we pop the top symbol C from the stack. The last character is *r* and the top element in the stack is Z_0, it indicates a read operation after a *if* statement, and thus we infer *lock downgrading* and push D into the stack. All the characters in the pattern have been visited and we are in the final state q_6, meaning the pattern is accepted by the automaton and a downgrading lock is inferred for later refactoring.

3.3 Transformation

FineLock conducts transformation on AST by converting synchronized locks into *ReentrantReadWriteLock* locks. In each refactored class, *FineLock* first imports the *ReentrantReadWriteLock* package and then defines the instance variable with the type *ReentrantReadWriteLock*. All locks are put into the *try...finally...* structure to ensure releasing a lock in case an exception is triggered. For downgrading lock, *FineLock* handles the transformation in the same way as the code shown in Figure 1(c). For splitting lock, *FineLock* performs the splitting operations for each read/write operation.

4 EVALUATION

We implement a prototype *FineLock* leveraging Wala [6], and the prototype is integrated as a plug-in of Eclipse, and we empirically evaluate it to address the following question:

RQ1: How effective and efficient is *FineLock* in refactoring coarse-grained locks into fine-grained locks?

RQ2: How useful is *FineLock* in improving throughput?

The evaluation was conducted on a HP Z240 workstation with 3.6 GHz Intel Core i7 CPU and 8GB main memory. The operating system is Ubuntu 16.04, and the JDK version is 8.

4.1 Effectiveness and Efficiency in Refactoring

We evaluate the effectiveness and efficiency of *FineLock* in five real-world applications including HSQLDB [5], Jenkins [7], Cassandra [1], JGroups [2], and SPECjbb2005 [3]. The size of the projects varies from 12,519 to 431,022 source lines of code (SLOC), and the time used to refactor each project is on average 27.6s, which clearly indicates our prototype is generally applicable to large projects.

The evaluation results are summarized in Table 1, which presents the projects (column 1), the numbers of synchronized locks in the original projects (column 2) and the data after refactoring (columns 3-9). Each synchronized lock in the original projects is refactored into one of the following locks: a downgrading lock (column 3), a splitting lock (column 4), a read lock (column 5) or a write lock (column 6). Downgrading locks and splitting locks compose fine-grained locks while read locks and write locks compose *ReentrantReadWriteLock* locks. Column 7 shows the ratio between the total number of fine-grained locks (columns 3-4) and the total number of synchronized locks (column 2) and column 8 shows the ratio between the total number of *ReentrantReadWriteLock* locks (columns 5-6) and the total number of synchronized locks (column 2). The last column gives the ratio between the total number of refactored locks and the total number of synchronized locks, which demonstrates that our tool can achieve 100% lock refactoring.

Table 1: Refactored Locks by *FineLock*

Benchmark	Original #LOCKS	Refactored Locks #DL	#SL	#RL	#WL	%FINE	%R/W	%RE
HSQLDB	684	6	39	109	530	7%	93%	100%
Jenkins	274	3	14	19	238	6%	94%	100%
Cassandra	239	2	24	39	174	11%	89%	100%
JGroups	179	5	33	28	113	21%	79%	100%
SPECjbb2005	170	1	2	38	129	2%	98%	100%
Total	1,546	17	112	233	1,184	8%	92%	100%

To further answer the question, we check whether the code after refactoring by *FineLock* still have the same behaviors as the original code by running test cases and manually checking. Specifically, we run the existing developer test cases of all 5 projects, and the percentage of passed test cases is 100%. In addition, we manually check all refactored locks. Each refactored lock is assigned to two developers to avoid bias. In the process, we manually check the following aspects: 1) if the refactoring changes the behaviors of original code; 2) if an inferred lock is correct; 3) if a lock is inserted to the right position; and 4) if a lock is used correctly. The manual results demonstrate that all our refactored locks are correct.

4.2 Improving Throughput

To answer RQ2, we run the concurrency test cases of the two projects *HSQLDB* and *SPECjbb2005* on both the original code and our refactored code. We separately run JDBCBench (a test program in *HSQLDB*) with 100k, 200k, 300k and 400k transactions on both original code and refactored code, and the comparison result is shown in Figure 4. From the chart, we can see that the transaction rate improvement is not obvious when the number of transactions is 100k, and the transaction rate is improved by about 15%, 19%, and 22% when the number of transactions reaches 200k, 300k, and 400k. The results clearly indicate our refactored code has better throughput. Due to space limitation, the results for *SPECjbb2005* is not shown. The reason we did not do the experiments for the other three projects is that they do not provide concurrency test cases.

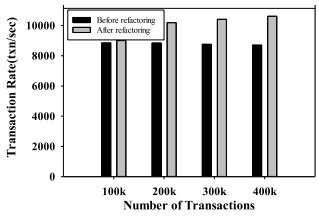

Figure 4: Throughput Comparison

5 CONCLUSIONS

This paper presents *FineLock*, an automatic refactoring tool to convert coarse-grained locks into fine-grained locks to reduce lock contention. *FineLock* is evaluated on 5 real-world applications. A total of 1,546 synchronized locks are refactored. Each project takes an average of 27.6 seconds. Experimental results show that *FineLock* can effectively refactor locks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was supported, in part by the SRF-Hebei ZD2019093, NSF-Hebei 18960106D and NSF-China 61802166.

REFERENCES

[1] Apache. 2019. *Cassandra.* https://cassandra.apache.org/
[2] Bela Ban. 2019. *JGroups.* http://www.jgroups.org/
[3] Standard Performance Evaluation Corporation. 2013. *SPECjbb2005.* https://www.spec.org/jbb2005/
[4] Eclipse. 2020. *Eclipse Java development tools.* https://www.eclipse.org/jdt/
[5] Hypersonic SQL Group. 2019. *HSQLDB - 100% Java Database.* http://hsqldb.org/
[6] IBM. 2018. *The t. j. watson libraries for analysis.* http://wala.sourceforge.net/wiki/index.php/Main_Pag
[7] Kohsuke Kawaguchi. 2019. *Jenkins.* https://jenkins.io/
[8] B. McCloskey, F. Zhou, D. Gay, and E. A. Brewer. 2006. Autolocker: synchronization inference for atomic sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* Association for Computing Machinery, 346–358.
[9] Benjamin Muskalla. 2008. *Concurrency-related refactorings for JDT.* https://wiki.eclipse.org/Concurrency-related_refactorings_for_JDT
[10] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. 2011. Refactoring Java programs for flexible locking. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 71–80.
[11] Sixth and Red River Software. 2007. *Locksmith.* https://plugins.jetbrains.com/plugin/1358-locksmith
[12] Y. Zhang, S. Dong, X. Zhang, H. Liu, and D. Zhang. 2019. Automated Refactoring for Stampedlock. *IEEE Access* 7, 1 (2019), 104900–104911.