International Journal of Grid and Distributed Computing
Vol. 9, No. 6 (2016), pp.71-82
http://dx.doi.org/10.14257/ijgdc.2016.9.6.08

Automated Refactoring from Synchronized Locks to Reentrant Locks

Yang Zhang*, Dongwen Zhang, Huiyong Wang

School of Information Science and Engineering,
Hebei University of Science and Technology, Shijiazhuang, Hebei, China
{zhangyang, zdwwtx, why}@hebust.edu.cn

Abstract

Multi-threaded Java applications using synchronized or reentrant locks respectively exist performance trade-off. Parallel programmers usually use manual refactoring to test the performance of locks in a particular environment. However, manual refactoring is labor-intensive and error-prone. There is a strong need for automated support to find which one is more suitable to exert the performance of applications. This paper presents a refactoring framework to enable the transformation from synchronized locks to reentrant locks automatically. The framework analyzes global monitors based on Quad intermediate representation and visitor pattern in the Joeq compiler, validates the consistency of analysis results, and performs the transformation. The framework is evaluated on three benchmarks including red-black tree, producer-consumer problem, and SPECjbb2005. The total time is less than 4s for all three benchmarks on common desktop computer, and the successful refactoring results are observed.

Keywords*: Refactoring, Synchronization, Reentrant lock, Software analysis*

1. Introduction

Lock is the common-used synchronization way to ensure correct program states in multi-threaded Java programs. Once a thread has already held the build-in monitor, it prevents other threads from accessing shared datum by turning shared state into private one. In multi-core era, lock contention is considered as a problem of lacking scalability. Transactional memory (TM) and lock-free algorithms are presented as alternatives. However, TM is not applicable to some cases (*e.g.* thread communications and I/O operations, *etc*) and lock-free algorithms are difficult to learn. Although many other synchronization mechanisms are presented, locks seem unlikely to be completely supplanted and will continue being used into the future.

In multi-threaded Java programming at early stage, synchronized blocks and methods are provided to programmers as the unique synchronization control way. It is the simple and easy-to-use ways for programmers to learn and understand. However, this synchronization way suffers from uninterrupted problems. For example, once one thread has locked a critical section, the others that are going to access the same critical section cannot go on executing until the thread release the lock. Since JDK 5, ReentrantLock is introduced as an extended implementation. It provides many interesting feathers, such as non-blocking structure, lock acquisition interruption, attempting to test whether a lock is held, and fairness.

Many programmers are facing with the problem: “Is the performance of programs using ReentrantLock absolutely superior to that using synchronized lock?”. We evaluate the performance of programs using synchronized and reentrant locks respectively and

* The corresponding author is Yang Zhang.

ISSN: 2005-4262 IJGDC
Copyright ⓒ 2016 SERSC

show that there is performance tradeoff between them for different applications (See Section 2).

```
public class SyncTest {
    private ArrayList<Integer> myList;

    public SyncTest(ArrayList<Integer> myList)
{
        this.myList = myList;
    }

    public synchronized Object get(int index) {
        return myList.get(index);
    }



    public synchronized boolean insert(int
newValue) {
        return myList.add((Integer)newValue);
    }


}
```

(a) Synchronized-based List

```
public class ReenTest{
    private ArrayList<Integer> myList;
    private Lock lock = new ReentrantLock();
    public ReenTest(ArrayList<Integer> myList){
        this.myList = myList;
    }

    public Object get(int index){
        lock.lock();
        try{
            return myList.get(index);
        }finally{ lock.unlock(); }
    }
    public boolean insert(int newValue) {
        lock.lock();
        try{
            return myList.add((Integer)newValue);
        }finally{lock.unlock(); }
    }
}
```

(b)ReentrantLock-based List

Figure 1. Source code of synchronized-based and Reentrant Lock-based Array Lists

Programmers usually use manual refactoring to transform from one lock to another. However, manual refactoring will be labor-intensive and error-prone because the transformation requires to find all the build-in monitors and to guarantee the consistency semantics. Furthermore, manual refactoring is unfeasible for legacy Java applications due to the lack of source codes. Therefore, there is a strong need for automated support to help programmers to learn about what works best in a particular application.

This paper presents a refactoring framework to transform Java applications from synchronized locks to reentrant locks automatically. The framework relies extensively on program analysis to keep the consistency of lock objects between lock mechanisms. The Quad intermediate representation in Joeq compiler [1] is used to facilitate the analysis of Java bytecode. By using visitor pattern analysis, all lock requests and thread communication operations are found. The consistency of lock sequence is validated based on the analysis results. The transformation of Java bytecode is implemented by the tool Javassist [2]. We implement it as a tool and evaluate on three benchmarks including red-black tree, producer-consumer problem, and SPECjbb2005. The experimental results show that all benchmarks are transformed successfully and the refactoring time is acceptable.

The contributions of this paper are as follows:

- Comparing the performance of synchronized locks with that of reentrant locks via four data structures,
- The detailed refactoring process of Java bytecode analysis and transformation,
- An implementation of a refactoring tool, Lock2Lock, automatically transforming synchronized locks to reentrant locks, and
- An evaluation of Lock2Lock on a set of Java applications, demonstrating its practicability.

The rest of this paper is organized as follows. Section 2 presents some examples to motivate the refactoring. Section 5 presents the detail of the refactoring including

Copyright © 2016 SERSC

analysis, validation and transformation. Section 7 presents the evaluation on a set of Java benchmarks. Related literatures are examined in Section 8 and conclusions are drawn in Section 9.

2. Why to Refactoring

This section presents some motivating examples that implemented by using synchronized and reentrant locks respectively. These examples focus on four general thread-unsafe data structures (HashMap, TreeSet, ArrayList, and LinkedList) that shared by numbers of read threads and write threads. Figure 1 shows the source code of synchronized-based and ReentrantLock-based ArrayList. The implementation of other three examples is similar to it. By using these examples and performance results, we illustrate that it is really a difficult work to choose the right lock mechanism for a specific application.

Figure 2 plots the execution time with the mutative scale of read threads and write threads. The experimental setup is given in Section 7.1. All measurements are averages of 10 runs. In Figure 2(a), the execution time of synchronized-based implementation is lower than that of ReentrantLock-based implementation under the situation of 10 RTs 90 WTs and 90 RTs 10 WTs. But the opposite conclusion can be drawn in Figure 2(d). For *LinkedList* benchmark, the performance of ReentrantLock-based implementation is better than that of synchronized-based implementation. For *TreeSet* and *ArrayList* benchmarks, the execution time highly depends on the scale of read and writes threads. The execution time of ReentrantLock-based implementation is much higher than that of synchronized- based implementation under the situation of 90 RTs 10 WTs for *TreeSet* bench- mark and 50 RTs 50 WTs and 90 RTs 10 WTs for *ArrayList* benchmark. Concluded from the experimental results, reentrant locks don't absolutely superior to synchronized lock although reentrant locks are the extended implementation. There is a performance tradeoff between them.

3. What to Refactoring

Table 1 shows the source code and bytecode [1] of two lock mechanisms respectively. For source code, ReentrantLock-based implementation commonly put the unlock operation into the structure try{ ... }finally{...} to ensure this operation to be always executed even if exceptions occur, while synchronized-based implementation does not need to do so apparently. Although different forms exist in source code, structures of bytecode are similar. Both have one lock operation and two unlock operations (for synchronized blocks, one *monitorenter* operation and two *monitorexit* operations). This enables us to make the replacement between them. For the *wait* and *notify* (or *notifyAll*) operations, we can make the replacement with *await* and *signal* (or *signalAll*) operations as well.

[1] All instructions are generated by the command javap -c classname

Copyright © 2016 SERSC

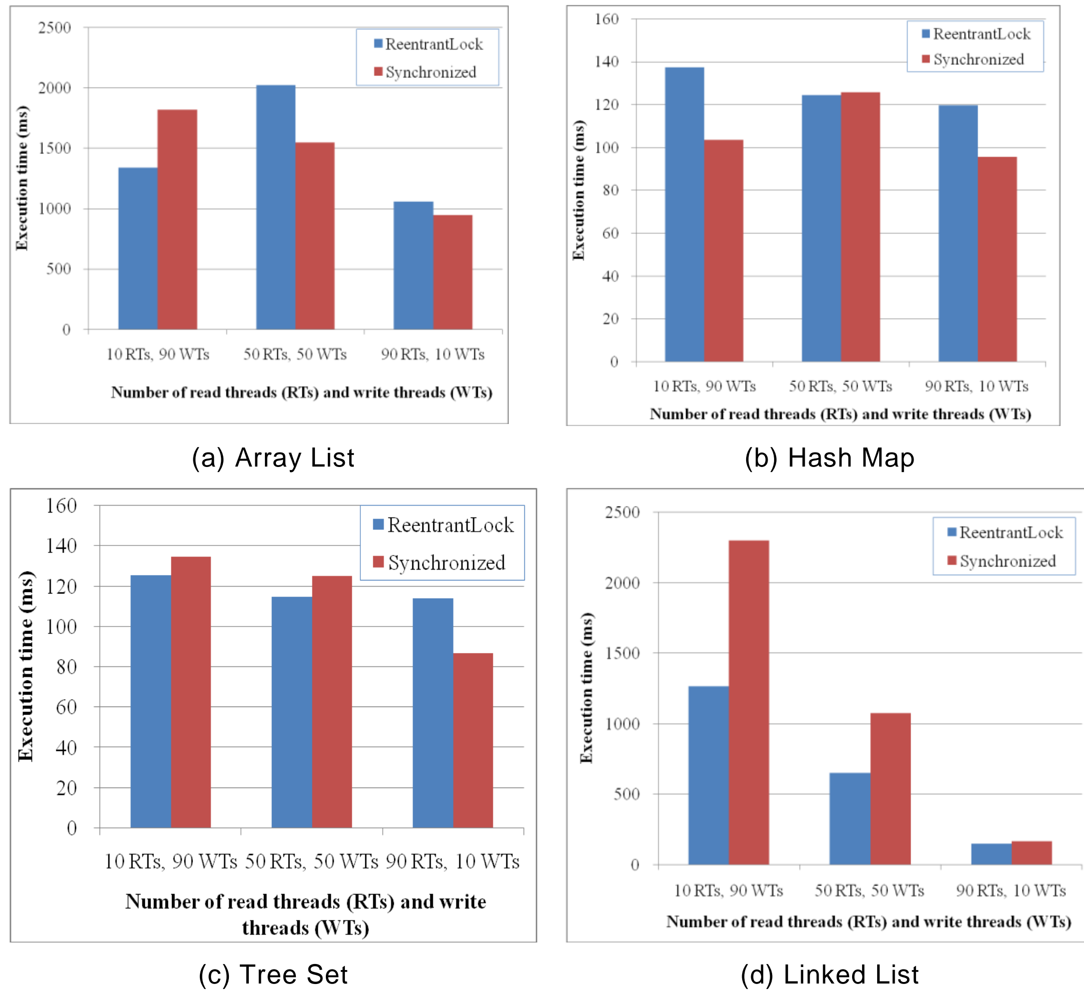


Figure 2. Performance of Java Applications using Synchronized or Reentrant Locks Respectively

4. Precondition

We need to guarantee that refactoring from synchronized locks to reentrant locks are semantics-preserving. It means that all the operations on the same build-in monitor will work on the same reentrant lock. The Java API Specification says that "a reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities" [3]. This ensures that the precondition will be satisfied.

To make the replacement semantic-preserve, the main problem is to how we ensure that all the synchronized operations on the current object this will work on the same reentrant lock *lock* in Table 1. Our solution for associating the current object this with the lock *lock* is to insert a new field to the class of the current object and to make this field public to allow access from any package. A set of locks L for each class is built to determine which locks will be inserted into this class.

Copyright © 2016 SERSC

Table 1. Source Code and Bytecode of Synchronized-based Implementation and ReentrantLock-based Implementation

Before refactoring	After refactoring
Source code	

synchronized(this){
...
}
	private Lock lock = new ReentrantLock();
	lock.lock();
	try{...
	}finally{ lock.unlock(); }

Bytecode	

0: aload 0
1: dup
2: astore 2
3: monitorenter
...
12: aload 2
13: monitorexit
14: goto 22
17: astore 3
18: aload 2
19: monitorexit
20: aload 3
21: athrow
22: return
	0: aload 0
	1: getfield #4; //Field lock:Ljava/util/concurrent/locks/Lock;
	4: invokeinterface #5, 1; //InterfaceMethod java/util/concurrent/locks/Lock.lock:()V
	...
	17: aload 0
	18: getfield #4; //Field lock:Ljava/util/concurrent/locks/Lock;
	21: invokeinterface #9, 1; //InterfaceMethod java/util/concurrent/locks/Lock.unlock:()V
	26: goto 41
	29: astore 2
	30: aload 0
	31: getfield #4; //Field lock:Ljava/util/concurrent/locks/Lock;
	34: invokeinterface #9, 1; //InterfaceMethod java/util/concurrent/locks/Lock.unlock:()V
	39: aload 2
	40: athrow
	41: return

5. How to Refactoring

This section presents how to refactor Java applications from synchronized locks to reentrant locks in details at the bytecode level.

5.1. Overview of Refactoring Framework

The overview of refactoring framework is shown in Figure 3. The whole process can be divided into several stages. First, Java bytecode intermediate representation (IR) is transformed into Quad IR, followed by program analysis within the Joeq compiler. Second, consistency is validated on the analysis results because the lock sequence of Quad IR cannot be always same with the lock sequence of bytecode IR. Finally, synchronized-based implementation is transformed into ReentrantLock-based one via the bytecode manipulation framework-Javassist.

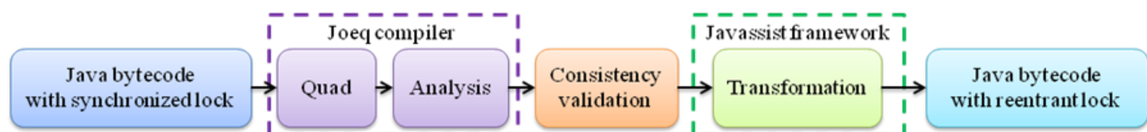


Figure 3. The Overview of Refactoring Framework

5.2. Analysis

Joeq compiler [1] is used to conduct the bytecode analysis during the refactoring. It translates Java bytecode IR into a three-addressed-like IR called Quad. Like Java bytecode, Quad retains program information such as field address and virtual method invocation, which makes the high-level optimization possible. Quad uses register-based architecture so that local variables and temporary information can be

Copyright © 2016 SERSC

reserved. This architecture is more conducive to program optimization than the stack architecture [1].

The refactoring framework uses visitor pattern analysis to find the position of synchronized locks and to get lock objects. Since synchronized methods and blocks have different forms, they are analyzed respectively.

To find the position of synchronized locks, the framework judges whether a method contains the synchronized modifier for synchronized methods. For synchronized blocks, the framework searches for the MONITORENTER and MONITOREXIT operators[2].

To get lock objects, for synchronized methods, the instance or static objects are obtained when instance and static methods are visited. For synchronized blocks, the framework analyzes the register name that followed the MONITORENTER and MONITOREXIT operators. According to the register name, the framework generates an unique name to represent each lock object and inserts the name as a new field into the corresponding class.

5.3. Consistency Validation

Each *monitorenter* and *monitorexit* instructions have the corresponding Quad instructions, such as “MONITORENTER” and “MONITOREXIT” followed by register name. However, the emerging sequence of Quad instructions is not always same with the sequence of build-in monitors in Java bytecode. Therefore, it is necessary to validate the consistency in order to perform correct transformation.

To validate the consistency, the most intuitive solution is to re-visit the original bytecode to ensure the correct sequence. However, it is a time-consumed solution and needs to find which *monitorenter* instruction corresponding to which “MON-ITORENTER” operator in Quad representation. The simple solution is to reuse the analysis results in which the visiting sequence of each “MONITOENTER” operator is recorded in the ordered lock list. Each “MONITOENTER” operator has a field of bytecode offset (implemented by the method getBCI() of the class joeq.Compiler.Quad.Quad). We check the field to guarantee the visiting sequence of “MONITORENTER” Quad in each class is emerged from low offset to high offset.

5.4. Transformation

The bytecode transformation tool Javassist [2] (version 3.13.0) is used to perfor-m the transformation. We firstly insert the lock fields to each class according to the elements of lock set, and then run the instance of class *Instrumentor* inherited from *javassist.expr.ExprEditor*. For synchronized blocks, the class *Instrumentor* overwrites the method edit for *monitorenter* and *monitorexit* to transform lock operations, and the method edit for *MethodCall* to transform the communication operations among threads. For synchronized methods, synchronized modifier is deleted from the method modifiers. Lock (or unlock) operations are inserted before (or after) methods. The ordered lock list for each class is used during the transformation. The elements of ordered lock list will be taken as the lock or unlock operation objects.

6. Implementation of the Refactoring Framework

We implement the refactoring framework as a library, named Lock2Lock. Lock2Lock will be downloaded at the web site http://code.google.com/p/lock2lock. The total size of the Lock2Lock library is less than 30KB of Java bytecode.

[2] The monitorenter and monitorexit operations in bytecode have the corresponding form of MONITORENTER and MONITOREXIT in Quad representation

 Copyright © 2016 SERSC

7. Experimentation

In this section, we report on an experimental evaluation of Lock2Lock on series of benchmarks.

7.1. Experimental Setup

All measurements are conducted on a quad-core 2.13GHz Intel Xeon processor running 64-bit Linux 2.6.38 kennel with 12GB RAM. Three benchmarks including red-black tree, producer-consumer problem, and SPECjbb2005 are selected to evaluate the refactoring.

7.2. Experimental Results

(1) Red-Black Tree

Red-Black Tree (RBTree) is a type of self-balancing binary search tree. RBTree is commonly seen as a standard benchmark when evaluating software transactional memory (STM). This benchmark is published by Herlihy *et al*. in dstm2 [4] with transactional read or write operations. We convert it to be a synchronized-based version by adding synchronized-modifier to those method in which transactional read (or write) operations exist. The total number of synchronized methods is 5.

Lock2Lock refactors RBTree successfully. The total execution time of Lock2Lock is 842 ms (analysis time is 654 ms, and transformation time is 188 ms).

The execution time against the scale of read threads and write threads is shown in Figure 4. The execution time of synchronized-based RBTree is less than that of ReentrantLock-based RBTree. No matter what kind of locks are considered, the execution time of either synchronized-based or ReentrantLock-based RBTree is almost same although the scale of read and write threads is changed.

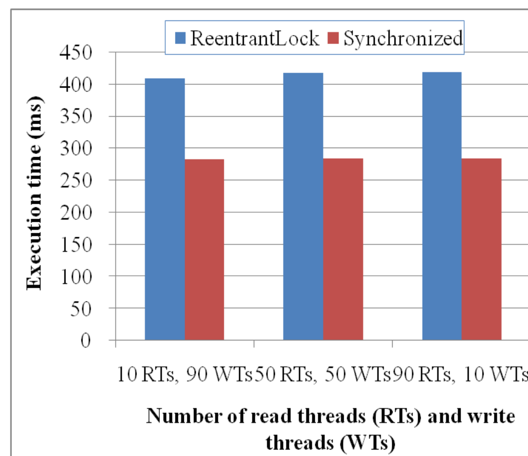


Figure 4. The Execution Time of Synchronized-based and ReentrantLock-based RBTrees

(2)Producer-consumer problem(PC problem)

PC problem is a classic problem that can be used to evaluate synchronization and collaboration among producer threads and consumer threads which share a fixed size buffer. Producer threads generate a piece of data once a time and put it into the shared buffer, while consumer threads get out the data from the shared buffer. The correct collaboration will make sure that producer threads will not put data into a full buffer and consumer threads will not get out the data from an empty buffer.

Copyright © 2016 SERSC

In our previous paper [5], we solve PC problem using aspect-oriented programming by separation of synchronization concern. Here we use its object-oriented version, not aspect-oriented version. This benchmark includes 2 synchronized blocks and 4 conditional operations.

Lock2Lock can refactor the code of PC problem successfully. Lock2Lock analyzes it firstly. After the bytecode analysis, PC problem do not pass the consistency validation. It means there are some inconsistent between build-in monitor object and lock object. The sequence of these objects is adjusted according to the methodology in Section 5.3 to ensure the exception *java.lang.IllegalMonitorStateException* will not happen. The total time of Lock2Lock is 608 ms (analysis time: 450 ms, transformation time: 158 ms).

The performance results are reported in Figure 5. The execution time of synchronized-based PC problem is almost same with that of ReentrantLock-based PC problem when the same number of read threads and write threads are considered.

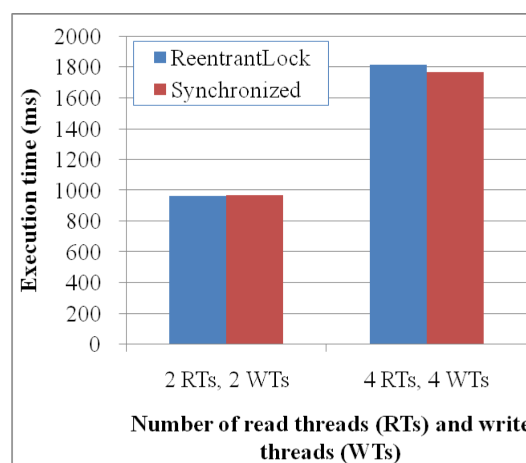


Figure 5. The Execution Time of Synchronized-based and ReentrantLock-based PC Problem

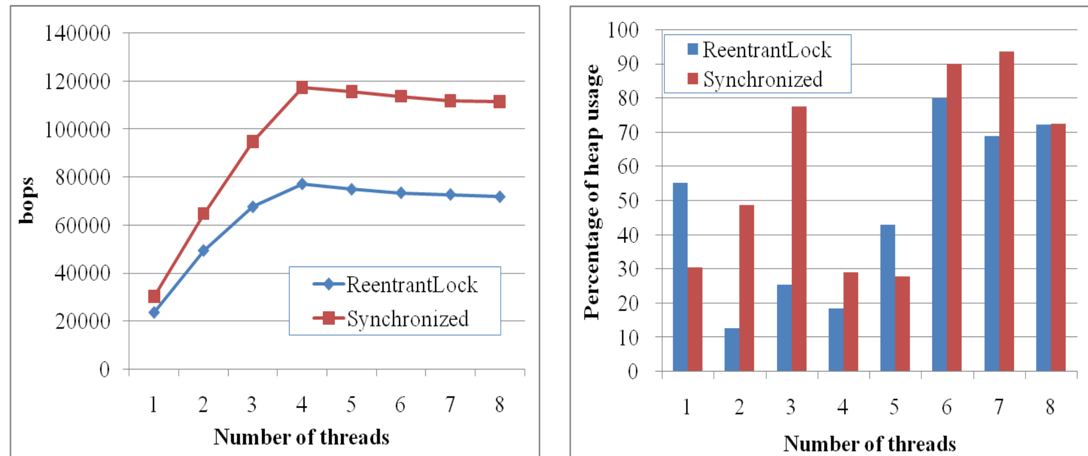
(3) SPECjbb2005

SPECjbb2005 is used as a stand-alone Java application emulating a 3-tier system with emphasis on the middle tier. SPECjbb2005 includes 165 synchronized methods and 22 synchronized blocks, as well as 8 conditional operations, so that it is thought as one of the best benchmarks to test the performance and correctness of Lock2Lock. The result shows Lock2Lock can refactor SPECjbb2005 successfully. The total execution time is 3610 ms (analysis time is 2362 ms, and transformation time is 1248 ms).

Figure 6 compares the performance of original Java implementation of SPECjbb2005 to that of the refactored implementations using Lock2Lock. It reports the number of bops, which is a measure of the number of transactions per second, and the percentage of heap memory usage against the number of threads. Figure 6(a) shows that synchronized-based implementation presents the better performance than ReentrantLock-based implementation, while Figure 6(b) proves that the percentage of heap memory usage in synchronized-based implementation is higher than that of ReentrantLock-based implementation under most of situations (6 out of 8 situations). The experimental results support our motivation that each lock has its advantages and disadvantages. IF the number of business operations is your concern, synchronized-based implementation should be used. Otherwise, ReentrantLock-based implementation should be adopted if the size of heap memory is your concern. Lock2Lock provides an alternative for programmer- s to transform

 Copyright © 2016 SERSC

from synchronized-based implementation to ReentrantLock-based implementation without breaking down the source code.



(a) Business Operations per Second (b) Percentage of Heap Memory Usage

Figure 6. SPECjbb2005 Performance

8. Related Works

In recent years, significant advances have been made in the area of automated tool support for refactoring. Early refactoring are mainly oriented to sequential programs [6][7][8]. Refactoring for concurrency [9][10][11][12] are becoming a research thrust with the ubiquity of multi-core processors in recent years. Lock2Lock can be considered as a refactoring tool to help the lock design of concurrent programs. Binary refactoring [13] is a technique for introducing optimizing transformation in object-oriented programs without affecting the program source code. Similar to binary refactoring, our refactoring is related to both bytecode analysis and transformation. These are large and diverse research areas, so our presentation is limited to closely related or representative works.

Relocker [14] is an automated tool that assists programmers with refactoring synchronized blocks into ReentrantLocks and ReadWriteLocks. Relocker has been implemented as a plug-in for Eclipse IDE to perform source-to-source transformation. Lock2Lock is different from Relocker in that 1) Lock2Lock is implemented by Joeq and Javassist while Relocker is based on WALA [15]. Different techniques are used to implement these two tools. 2) Lock2Lock performs bytecode transformation without modifying the source code. Lock2Lock takes the refactoring from synchronized lock to read-write lock as its future work.

Dig *et al.* [10] refactor sequential programs to be reentrant, parallel ones using the *java.util.concurrent* utilities. It makes shared data accesses thread-safe by converting *int* to *AtomicInteger* and converting *HashMap* to *ConcurrentHashMap*. Wloka *et al.* [16] present a refactoring tool that makes single-threaded programs reentrant by replacing global state with thread-local state and performing each execution in a fresh thread. Lock2Lock highlights the separation of synchronized concerns. Our work highlights lock refactoring while their works concern about parallelization of code.

FlexSync [17], an aspect-oriented synchronization library, enables customization of multiple synchronization mechanisms (*e.g.* lock, atomic block, and STM). The limitation of FlexSync is that their lock implementation does not handle ReentrantLock in the Java 5 library. Our implementation Lock2Lock can be taken as a supplement to FlexSync. Moreover, FlexSync performs not well in handling

Copyright © 2016 SERSC

wait/notify semantics due to the limitation of STM while Lock2Lock has no such limitation.

SyncGen [18] is a general tool that synthesizes complex synchronization implementation. It separates the synchronization and functional implementation, and provides aspect-oriented support for weaving them together. Autolocker [19] is a lock inference tool. Locks are acquired before object accesses and released at the end of the atomic section. Different from Autolocker, Lock2Lock do not infer the lock, but transform to a new lock mechanism based on existing synchronized-based lock mechanism.

9. Conclusions

This paper presents a refactoring that makes the programs using reentrant locks by replacing synchronized locks. The refactoring enables programmers to perform performance tradeoffs between these two locks. The refactoring is implemented in a tool called Lock2Lock which automatically accomplished the refactoring. We use Lock2Lock to refactor several synchronized-based Java applications and observe the successful refactoring results. The time of refactoring is within 4s even for large server applications SPECjbb2005 on common desktop computer.

Future work includes the refactoring to ReadWriteLock and atomic operations. We will continue to evaluate Lock2Lock on more benchmarks.

Acknowledgments

This work is partially supported by National Nature Science Foundation of China under grant No.61440012 and No.61300120, the top-notch young talent Foundation of Hebei Province of China under Grant No. BJ2014023, and Nature Science Foundation of Hebei Province under Grant No.F2012208016 and No.F2016208007. The authors also gratefully acknowledge the insightful comments and suggestions of the reviewers, which have improved the presentation.

References

[1] J. Whaley, "Joeq: A virtual machine and compiler infrastructure", Science of Computer Programming vol.57, no.3, **(2005)**, pp. 339–356.

[2] S. Chiba, "Javassist - a reflection-based programming wizard for java", Proceedings of OOPSLA 98 Workshop on Reflective Programming in C++ and Java, Denver, America, **(1998)**, pp. 92–115.

[3] Oracle, "java.util.concurrent.locks api specification". http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/package-summary.html. (2014) Last accessed **(2014)**, pp. 1-27.

[4] M. Herlihy, V. Luchangco, M. Moir, "A flexible framework for implementing software trans- actional memory", ACM SIGPLAN Notices, Vol. 41, no. 4, **(2006)**, pp. 253–262.

[5] Y. Zhang, J. Zhang, D. Zhang, "Implementing and testing producer-consumer problem using aspect-oriented programming", Proceedings of Fifth International Conference on Information Assurance and Security (IAS), Xi'an, China, IEEE, **(2009)**, pp. 749–752.

[6] H. Kegel, F. Steimann, "Systematically refactoring inheritance to delegation in java", Proceedings of the 30th international conference on Software engineering, Leipzig, Germany, **(2008)**, pp. 431–440.

[7] M. Kiezun, M. Ernst, F. Tip, R. Fuhrer, "Refactoring for parameterizing java classes", Proceedings of 29th International Conference on Software Engineering, Minneapolis, America, **(2007)**, pp. 437–446.

[8] F. Steimann, A. Thies, "From public to private to absent: Refactoring java programs under constrained accessibility", Proceedings of ECOOP 2009–Object-Oriented Programming, **(2009)**, pp. 419–443.

[9] S. Markstrum, R. Fuhrer, T. Millstein, Towards concurrency refactoring for x10, ACM Sigplan Notices, Vol. 44, ACM, **(2009)**, pp. 303–304.

[10] D. Dig, J. Marrero, M. Ernst, Refactoring sequential java code for concurrency via con- current libraries, Proceedings of IEEE 31st International Conference on Software Engineering (ICSE), IEEE, **(2009)**, pp. 397–407.

[11] D. Dig, M. Tarce, C. Radoi, M. Minea, R. Johnson, "Relooper: refactoring for loop paral- lelism in java", Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, ACM, **(2009)**, pp. 793–794.

 Copyright © 2016 SERSC

[12] Y. Zhang, W. Ji, “A scalable method-level parallel library and its improvement”, The Journal of Supercomputing. vol. 61, no 3. **(2012)**, pp. 1154–1167.
[13] E. Tilevich, Y. Smaragdakis, Binary refactoring: improving code behind the scenes, in: Proceedings of the 27th international conference on Software engineering, ACM, **(2005)**, pp. 264–273.
[14] M. Schafer, M. Sridharan, J. Dolby, F. Tip, Refactoring java programs for flexible locking, Proceedings of 33rd International Conference on Software Engineering (ICSE), IEEE, **(2011)**, pp. 71–80.
[15] IBM, T.j watson libraries for analysis (wala) http://wala.sourceforge.net. Last accessed, **(2013)**, pp. 1-27.
[16] J. Wloka, M. Sridharan, F. Tip, “Refactoring for reentrancy”, Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, **(2009)**, pp. 173–182.
[17] C. Zhang, “Flexsync: An aspect-oriented approach to java synchronization”, Proceedings of the 31st International Conference on Software Engineering, Vancouver, Canada, IEEE Computer Society, **(2009)**, pp. 375–385.
[18] X. Deng, M. Dwyer, J. Hatcliff, M. Mizuno, “Syncgen: An aspect-oriented framework for synchronization”, Tools and Algorithms for the Construction and Analysis of Systems. Vol 14, no 9, **(2004)**, pp. 158–162.
[19] B. McCloskey, F. Zhou, D. Gay, E. Brewer, “Autolocker: synchronization inference for atomic sections”, ACM SIGPLAN Notices, Vol. 41, ACM, **(2006)**, pp. 346–358.

Authors

Yang Zhang was born in 1980, and received his PHD degree in School of Computer at Beijing Institute of Technology. He is an associate professor in school of Information Science and Engineering at Hebei University of Science and Technology. His research interests focus on parallel programming model and software refactoring for parallelism.

Dongwen Zhang was born in 1964, and received his PHD degree in School of Computer at Beijing Institute of Technology. He is an professor in school of Information Science and Engineering at Hebei University of Science and Technology. His research interests focus on parallel programming model and software refactoring for parallelism.

Huiyong Wang was born in 1980, and received his PHD degree in School of Computer at Hebei University of Technology. He is an assisstant professor in school of Information Science and Engineering at Hebei University of Science and Technology. His research interests focus on parallel programming model.

Copyright © 2016 SERSC

Copyright © 2016 SERSC