Refactoring Java Concurrent Programs Based on Synchronization Requirement Analysis

Binxian Tao  Ju Qian
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics Nanjing, China, 210016 
{taobinxian, jqian}@nuaa.edu.cn

Abstract— Writing high quality concurrent programs is challenging. A concurrent program that is not well-written may suffer from coarse synchronization problems, e.g., overly-large critical sections, overly-coarse locks, and etc. These coarse synchronizations may introduce unnecessary lock contention and thereby affect the parallel execution of running threads. To optimize them, people suggest use refactorings, e.g., Split Lock refactoring and Split Critical Section refactoring, to gradually evolve the synchronization code for better parallelism. However, manually identifying the refactoring opportunities is difficult and by-hand code transformations are error-prone. To reduce the manual efforts, this paper proposes an automated refactoring approach for Java concurrent programs based on synchronization requirement analysis. It can automatically analyze the existing synchronization code to identify synchronization requirements. Bases on these requirements, we can find Split Lock, Split Critical Section, and Convert to Atomic refactoring opportunities and then make proper code transformation for each of them. Our experiment shows that the approach does find effective refactoring opportunities in real projects and can transform the refactorable code correctly. This indicates the approach could be helpful for concurrent program evolution.

Keywords—Java; concurrency; synchronization; refactoring

I. INTRODUCTION

Concurrent programming is essential for modern software development. However, writing concurrent programs is challenging [1]. It is difficult for a programmer to write high quality concurrent code at one time. A concurrent program that is not well-written may suffer from coarse synchronization problems. Such coarse synchronizations can be due to unnecessarily large critical sections that protect irrelevant computations, critical sections that use single big locks to protect multiple groups of shared data which do not really need to be synchronized, or else. They introduce unnecessary lock contention, forcing threads spending more time in waiting, and thereby may affect the parallel execution of running threads.

For those coarse synchronizations, a possible solution could be using compilers to optimize them [2-6]. The compiler optimizations need no modification on the source code. However, they usually have strict requirements on safety and efficiency. This limits the techniques that could be used, and, therefore, we cannot depend on them to do all the optimizations. To be more optimized, researchers suggest that we could use source-code level refactorings, e.g., Split Lock, Split Critical Section, and Shrink Critical Section, to gradually evolve the concurrent code and achieve fine-grained synchronizations [7-12]. The refactoring approaches are promising and have fewer limitations. But they highly depend on the programmers’ experience in order to find proper refactoring opportunities, and by-hand code transformations are error-prone. To provide better supports for the evolution of concurrent programs, automated refactoring approaches are demanded. In the existing work, Greenhouse et al. [8] and LockSmith [11] provide automated supports for synchronization code refactoring. However, they still largely demand on user interactions. Greenhouse’s approach relies on user-provided annotations, while LockSmith needs the users to identify valuable refactoring targets. Using these approaches to do refactoring still needs lots of manual efforts.

To reduce the need for user interactions, in this paper, we propose an automated refactoring approach for Java concurrent programs based on synchronization requirement analysis. The approach supports Split Lock, Split Critical Section, and Convert to Atomic refactorings. It firstly infers the synchronization requirements between class fields according to their presence in the existing critical sections. Having known the synchronization requirements, we search for the refactoring opportunities and transform the found coarse-grained synchronization code into fine-grained one. In the analysis, if two groups of fields have no synchronization requirement and there is no special program structure forcing them to use the same protection locks, we transform their relevant critical sections into ones protected by different locks. If two parts of a critical section access shared data that need not to be synchronized, we split the critical section into two separate ones. For an integer or boolean typed field, if it has no synchronization requirement with other fields and the operations on it satisfy some given conditions, we convert the field into an atomic-typed one. The whole approach was implemented as an Eclipse plugin. We tested the plugin on several open source programs. The experimental results indicate that the approach does find effective refactoring candidates in real projects and can transform the source code correctly.

The rest of this paper is organized as follows. Section II presents two motivating examples for concurrent program refactoring. Section III introduces our synchronization requirement analysis algorithm. Section IV presents the refactoring methods based on synchronization requirement information. Section V discusses the implementation of our tool.
and the experimental study. Finally, we show the related work and conclude the paper in Section VI and VII.

## II. MOTIVATING EXAMPLES

Fig. 1 demonstrates an example for Split Lock refactoring. The example is extended from Lea’s famous book [9]. For easy discussion, we add a constructor and two methods `rotate` and `getPerimeter` to the original code. In the example, all the long calculation methods named `longCalculation*` are assumed to only access thread-local resources, and the location properties `{x, y}` and the dimension properties `{width, height}` are assumed unnecessary to be synchronized. In Fig. 1(a), fields `x`, `y`, `width`, and `height` are coarsely protected by the same lock in the critical sections. For a `Shape` object `o`, when one thread `A` is executing method `o.adjustLocation()`, if another thread `B` wants to call method `o.adjustDimensions()` to adjust the shape dimension, it has to wait a long time until the long calculation in method `adjustLocation` finishes and the lock held by thread `A` is released. However, the waiting is unnecessary, since there is no synchronization requirement between the location properties and the dimension properties and there is no special program structure forcing these two groups of properties to use the same protection lock. In Fig. 1(b), we apply Split Lock refactoring to the original code. After refactoring, the accesses to location properties and dimension properties are protected by two different locks, `locationLock` and `dimensionLock`, respectively. Now a thread can simultaneously call method `adjustDimensions` while another thread is executing method `adjustLocation`. By Split Lock refactoring, we can avoid unnecessary lock contention and make the program execute more concurrently.

Fig. 2 illustrates an example for Split Critical Section refactoring. In the figure, also assume that all the methods named `longCalculation*` only access thread-local resources and the location properties and dimension properties do not need to keep consistent by synchronizations. In Fig. 2(a), the accesses to fields `x`, `y`, `width`, and `height` are put into the same critical section, which causes the lock being held by method `adjust` for an overly long time. During the execution of method `adjust`, other threads who want to read the states of `x` and `y` need to wait for the end of `adjust`. Instead of statement `S1`, this waiting may reduce the parallelism of threads. However, it is actually unnecessary since field groups `{x, y}` and `{width, height}` do not need to be synchronized. In Fig. 2(b), we use

```java
class Shape {  
protected double width;  
protected double height;  
protected double x;  
protected double y;  
public Shape() {  
x = y = 0.0;  
width = height = 0.0;  
}  
public synchronized double x() { return x; }  
public synchronized double y() { return y; }  
public synchronized double width() { return width; }  
public synchronized double height() { return height; }  
public synchronized void adjustLocation() {  
x = longCalculation1();  
y = longCalculation2();  
}  
public synchronized void adjustDimensions() {  
width = longCalculation3();  
height = longCalculation4();  
}  
public synchronized double getPerimeter() {  
return (width + height) * 2;  
}  
public synchronized void rotate(double angle) {  
x = x * cos(angle) – y * sin(angle);  
y = x * sin(angle) + y * cos(angle)  
}  
}  

(a) before refactoring

(b) after refactoring

Fig. 1. An example for Split Lock refactoring

362
For programs with arbitrary time, we believe the results are extremely difficult. In practice, automated synchronization code refactoring makes it possible to refine synchronization requirements between them. Otherwise, these values may come from different time points, and their combination may become meaningless. In Java, programmers usually ensure synchronization requirements by synchronized methods or blocks.

To determine the synchronization requirements, we firstly introduce a fundamental assumption before analysis. Then, the detailed determination rules will be presented.

### A. Assumption for Synchronization Requirement Analysis

As discussed in section II, a key to determine synchronization requirements is to identify whether two fields are protected together in a critical section. For programs with arbitrary structures, the unclear relationships between locks and their protected data make such identification extremely difficult. For a critical section in a library module using under open environments, it is difficult to know whether an object accessed inside it can escape out of the enclosing thread. Even an object escapes out, it is difficult to know whether the object can possibly be shared between threads. For a shared object, whether it is protected by the current lock or other locking or non-locking structures is also hard to be determined. Conservatively assuming that all the possibly shared data accessed inside a critical section are protected by the lock is safe but may be too rough for practical use. Based on such assumption, we may enable to get useful analysis results for automated refactoring. To be more practical, we make a fundamental assumption for the concurrent code and analyze the synchronization requirements under this assumption.

**Assumption:** In a program for refactoring, instance fields are only protected by critical sections with their enclosing objects as the locks, when necessary in an multi-threaded environment.

According to the above assumption, for a critical section protected by a lock object \( O \), only the fields of \( O \), like \( O.f \), are considered being protected by it. For the example in Fig. 3, we consider that the lock in synchronized method \( copyTo \) only protects the accesses to fields \( this.x \) and \( this.y \); while the lock in method \( copyFrom \) only protects the accesses to fields \( from.x \) and \( from.y \).

The research assumption is not absolutely safe and there can be situations violating the assumption. For example, in Fig. 1(b), we can use object \( locationLock \) to protect instance fields \( x \) and \( y \) of another object. Even though, since the assumption is held in a lot of places, we believe the results obtained under it can still be helpful for the evolution of concurrent code. Based on these results, we may find more useful refactoring opportunities. This will reduce the manual efforts in refactoring candidate searching. In combination with user confirmations, the risk of using such assumption can be controlled.
Therefore, we will obtain \( (a) \). Then, there is a CoSync \( a \ b \) written in critical sections.

In critical sections without access to synchronization, the denominator is 0.)

Consider a method that protects both notions named protected together in critical sections, and there is a synchronization requirement as long as two fields are protected.

If the predefined threshold is 0.3. Unless the synchronization strength between any two fields does not occur together in any normal method but are exclusively written in critical sections in method adjustLocation, adjustDimensions, and rotate. Therefore, CoOccur \( (x, \ y) = 0 \), and ExclusiveWrite \( (x, \ width) = 3 \), and adjustLocation, height, width. Finally, we can get \( \text{SyncStrength}(x, \ width) = \frac{1}{(1 + 0 + 3)} = 0.25 \). Also suppose the predefined threshold is 0.3. Under this setting, there are still synchronization requirements between the fields inside each group. But the synchronization strength between any two fields in these two different groups is less than the predefined threshold. Therefore, there is considered to be no synchronization requirement between these two groups of fields.

### C. The Analysis Algorithm

Algorithm 1 presents the basic structure of the synchronization requirement analysis algorithm. The algorithm scans each method and analyzes the fields used inside it to determine the CoSync, CoOccur, andExclusiveWrite relationships. For a critical section, we categorize the protected fields into protected reads and protected writes, i.e., \( F_{\text{protected read}} \) and \( F_{\text{protected writes}} \) respectively. For a field access \( v \ f \), if the base object \( v \) could possibly be the lock object, namely there is a potential alias relationship \( 13 \) between variable \( v \) and the lock object (MayAlias \( (v, \ lock(S)) \)), then according to the assumption in Section III.A, we consider field \( f \) being protected by synchronization and it will be put into the protected field sets \( F_{\text{protected read}} \) or \( F_{\text{protected writes}} \). After identifying all the protected fields, the CoSync and ExclusiveWrite sets can be determined. Having scanned all the methods, the synchronization strength between every two fields \( f_1 \) and \( f_2 \) can be computed. If the strength is larger than the given threshold, these two fields will be considered having a synchronization requirement, and a field pair \( (f_1, f_2) \) will be put into the synchronization requirement set \( SyncReq \).

The algorithm needs to determine whether a variable is aliased to the lock object. Such aliases are not easy to be analyzed both safely and precisely, especially for the library code working on unknown contexts. To be safe, we adopt type-based alias analysis to find the alias relationships. For two variables \( v_1 \) and \( v_2 \), suppose their declaration types are \( T_i \) and \( T_j \), respectively. If there is an intersection between the concrete subtypes of \( T_i \) (the set of types which \( v_1 \) may bind at runtime). The synchronization strength will be the following. \((\text{SyncStrength}(a, b) = \frac{CoOccur(a, b) + \text{ExclusiveWrite}(a, b)}{CoSync(a, b)}\)
Algorithm 1: IdentifySyncRequirements

```
Input:  Program P
Output: The set of synchronization requirements: SyncReq
begin
  let lock(S) be the lock protecting critical section S;
  let Read(S) ∪ Write(S) be the instance field reads/writes in critical section S;
  let Accessed(M) be the set of all fields accessed by M;
  let THRESHOLD be the threshold for synchronization requirement analysis;
  foreach method M in P do
    Fprotected := ∅;
    foreach critical section S in M do
      Fprotected_read := ∅; Fprotected_write := ∅;
      foreach v ∈ Read(S) ∪ Write(S) do
        if MayAlias(v, lock(S)) then
          if v ∈ Write(S) then
            Fprotected_write := Fprotected_write ∪ {f};
          else
            Fprotected_read := Fprotected_read ∪ {f};
        end
      end
    end
    foreach f1, f2 in Fprotected_read ∪ Fprotected_write do
      if CoSync(f1, f2) then CoSync(f1, f2) := CoSync(f1, f2) ∪ {M};
    end
    foreach f1 ∈ Fprotected_write ∧ f2 ∈ Accessed(M) do
      if ExclusiveWrite(f1, f2) := ExclusiveWrite(f1, f2) ∪ {M};
    end
    Fprotected := Fprotected ∪ Fprotected_read ∪ Fprotected_write;
    if M is not a constructor then
      foreach f1, f2 in Accessed(M) do
        if ¬(f1 ∈ Fprotected ∧ f2 ∈ Fprotected) then
          CoOccur(f1, f2) := CoOccur(f1, f2) ∪ {M};
        end
      end
    end
    foreach field pair (f1, f2) do
      if CoSync(f1, f2) > 0 then
        SyncStrength(f1, f2) := (CoSync(f1, f2) + ExclusiveWrite(f1, f2));
      end
      if SyncStrength(f1, f2) = 0 then
        SyncReq := SyncReq ∪ {f1, f2};
      end
    end
  end
end
```

and the concrete subtypes of T2 (the set of types which v2 may bind at runtime), we consider that variable v1 and v2 may access the same objects and there is a potential alias relationship between them.

IV. CONCURRENT CODE REFACCTORING BASED ON SYNCHRONIZATION REQUIREMENT INFORMATION

We automate the Split Lock, Split Critical Section, and Convert to Atomic refactorings based on the synchronization requirement information.

A. Split Lock Refactoring

To conduct Split Lock refactoring, we need to classify class fields into partitions which can then be protected by separated locks. Obviously, to be behavior-preserving, any two fields with synchronization requirements should be put into the same partition. Besides, if two fields are protected by the same lock in some existing critical section before refactoring, we also need to put them into the same partition. Otherwise, it will be difficult to choose new locks for that critical section during refactoring, since the protected fields inside it occur in multiple partitions. According to these rules, finally we will get partitions of all fields PF = {F1, F2, ..., Fn}.

For a class C, if there exist two of its declared instance fields in different partitions and each declared field of class C does not occur in the same partition as fields declared in other classes, then the class will be considered as a candidate refactorable class. More specifically, let DeclFields(C) be the set of instance fields declared in class C. Then, a candidate class C should satisfy the following conditions:

\[ \exists f_1, f_2 \in \text{DeclFields}(C) \{ f_1 \in F_i, f_2 \in F_j, i \neq j \} \]
\[ \land \neg \exists f_1 \in \text{DeclFields}(C), f_2 \in \text{DeclFields}(C') \{ f_1 \in F_i \land f_2 \in F_k \} \]

For each candidate refactorable class C, we then search for a detailed lock splitting strategy. Before searching, all the critical sections protecting class C’s instance fields are found. If all the lock objects of these critical sections are of declared type C, then class C can be selected for Split Lock refactoring. For each field partition F of class C, we check whether there are \text{wait()}/\text{notify()} calls on the lock objects in the critical sections protecting F. If such calls exist, then the lock of partition F cannot be replaced, since the original lock is required to protect the \text{wait()}/\text{notify()} calls. If no such calls exist, we can introduce a new separate lock for partition F and then replace the old lock which protects F with the new lock. As shown in Fig. 1(b), the new locks are declared as \text{java.lang.Object typed final instance fields of the class. They are} firstly named as \text{lock1, lock2, ..., lockn}. The users can then rename them for more meaningful names.

In the example of Fig. 1, the fields of class \text{Shape} can be classified into two partitions \{x, y\} and \{width, height\} according to the synchronization requirements between them and their occurrences in the critical sections. Since all the lock objects of the critical sections protecting these fields are of type \text{Shape}, the refactoring conditions can be satisfied and there is a Split Lock refactoring opportunity. Because there is no \text{wait()}/\text{notify()} call in the critical sections, we can introduce a new lock for each field partition of class \text{Shape}. Then, the Split Lock refactoring can be conducted by replacing the old lock with new locks. After refactoring, the original synchronized methods will be replaced by a set of synchronized blocks (see Fig. 1(b)). For a previously existing synchronized block \text{synchronized(s.locationLock)\{ ... \}}, it will be transformed into a form like \text{synchronized(s.locationLock)\{ ... \}}, where the old lock \text{s} will be replaced by the newly introduced lock field \text{s.locationLock}.

In the after-refactoring critical sections, the type of the lock objects will be \text{Object}. This type is not the same as that of the protected objects. Therefore, the Split Lock refactoring conditions will not be satisfied again on the new critical sections. This prevents our tool from refactoring critical sections repeatedly and generating incorrect results.
B. Split Critical Section Refactoring

For Split Critical Section refactoring, we analyze each critical section to determine whether it can be split into separate parts and make appropriate code transformation suggestions for the user. The key steps that find the splittable critical sections are demonstrated in Algorithm 2. The algorithm checks each line in a critical section to find the splittable positions. If the splittable position set is not empty, the critical section will be considered splittable. For a line \( l \) in the critical section, we analyze the synchronization requirements between the fields protected before \( l \), i.e., Before\( l \), and the fields protected after \( l \), i.e., After\( l \). If there is no synchronization requirement between these two parts of fields, we consider that the critical section can be split into two new separated ones at line \( l \). The line will be added to the candidate splitting position set \( \text{Locations} \) and reported to the user.

For method \( \text{adjust} \) in Fig. 2, the algorithm checks each potential position to determine whether the critical section can be split there. When encountering statement \( S_i \), we can find the field protected before and after \( S_i \) are \( \{x, y\} \) and \( \{\text{width}, \text{height}\} \), respectively. As discussed in section III.B, when setting the synchronization strength threshold to 0.3, there is no synchronization requirement between these two groups of fields. Consequently, the critical section can be split at \( S_i \). The after-refactoring critical section is shown in Fig. 2(b).

Algorithm 2 does not show how the \( \text{wait}() \)/\( \text{notify}() \) calls are handled. For a critical section with such calls, we require the refactoring do not break the continuous synchronization protection on a \( \text{notify}() \) call and its following statements. These following statements are guaranteed to be “seen” by a corresponding \( \text{wait}() \) operation waiting on the lock of the critical section. If we break the synchronization, there will be no such guarantee, and hence the transformation can be dangerous. For other situations, we believe splitting the \( \text{wait}() \)/\( \text{notify}() \) calls will not affect the program behaviors. Therefore, these calls will be handled as normal statements.

C. Convert to Atomic Refactoring

If a field uniquely protected by a lock has type \( \text{int} \), \( \text{long} \), or \( \text{bool} \), and the operations on the field match to the methods of atomic types such as \( \text{AtomicInteger} \), \( \text{AtomicLong} \), or \( \text{AtomicBoolean} \), then the field and its corresponding synchronizations can be replaced by atomic type based implementations. Fig. 4 presents an example for such Convert to Atomic refactoring. In Fig. 4(a), suppose there is only one critical section \( \text{inc} \). Then, the integer field \( a \) is uniquely protected by the implicit lock \( \text{this} \) in the critical section, and the increment computation on \( a \) matches to method \( \text{getAndIncrement} \) in \( \text{AtomicInteger} \). Therefore, the preconditions of Convert to Atomic refactoring are satisfied. We can transform the field and the synchronizations on it to \( \text{AtomicInteger} \) based implementation. The code after refactoring is shown in Fig. 4(b). The refactoring makes the source code simpler and cleaner, so that it can be read and maintained more easily. Under some platforms, using atomic type based implementations also may improve the performance of the program.

In our approach, we scan for Convert to Atomic refactoring opportunities after the Split Lock refactoring. Having done Split Lock refactoring, class fields with no synchronization requirement to others tend to be protected by separate lock objects. Based on such code, more Convert to Atomic refactoring opportunities can be found. The scanning of the code satisfying the preconditions of Convert to Atomic refactoring is trivial, and hence we omit its details here.

V. Experiments

To validate the proposed approach, we implemented it as an Eclipse plugin and evaluated it on several open source projects. The implementation is based on JDT (Java Development Tools) [14] and Soot bytecode analysis infrastructure (2.4.0) [15]. In the plugin, we connect JDT and Soot via entity names and source code line information. The tool firstly finds the critical sections in the source code with JDT and then analyzes all their relevant code at bytecode level with Soot. At the bytecode level, the synchronization requirements are identified and the refactoring candidates are collected. Then, we map the bytecode-level analysis results to source code level, and the refactoring transformations can be conducted at source code level with JDT.

In the implementation, the effects of method calls are also considered, and both the directly and indirectly accessed fields are counted. But when calculating \( \text{CoSync} \) and \( \text{ExclusiveWrite} \),

```
Algorithm 2: SplitCriticalSection

Input: program P
Output: \( \text{SplitCandidates} \), the splittable critical sections together with the locations where they can be split

begin
  let \( \text{Sync}(P) \) be the set of all synchronized methods or blocks in \( P \);
  let \( \text{Line}(S) \) be the lines of statements in a critical section \( S \);
  let \( \text{SyncReq} \) be synchronization requirement set (field pairs);
  let \( \text{Before}(l) \) be the fields protected by the critical section before line \( l \);
  let \( \text{After}(l) \) be the fields protected by the critical section after line \( l \)
    (include \( l \) itself);

  foreach \( S \in \text{Sync}(P) \) do
    \( \text{Locations} := \emptyset \);
    for \( l \in \text{Line}(S) \) do
      if \( \forall f \in \text{Before}(l), f \notin \text{Sync}(S) \) then
        \( \text{Locations} := \text{Locations} \cup \{1\} \);
      end
    end
    if \( \text{Locations} \neq \emptyset \) then
      \( \text{SplitCandidates} := \text{SplitCandidates} \cup \{S, \text{Locations}\} \);
    end
  end
end
```

\[
\begin{array}{ll}
\text{class Test} \{ \\
\quad \text{int } a; \\
\quad \text{AtomicInteger } \text{a} = \text{new AtomicInteger}(); \\
\quad \text{public void synchronized inc}\{ \\
\quad \quad a++; \\
\quad \} \\
\} \\
\end{array}
\]

(a) before refactoring

\[
\begin{array}{ll}
\text{class Test} \{ \\
\quad \text{AtomicInteger } \text{a} = \text{new AtomicInteger}(); \\
\quad \text{public void inc}\{ \\
\quad \quad a.getAndIncrement(); \\
\quad \} \\
\} \\
\end{array}
\]

(b) after refactoring

Fig. 4. An example for Convert to Atomic refactoring
only methods directly containing critical sections are taken into account. When calculating CoOccur(a, b), we do not count a method if it accesses two fields a and b only because one of its callee accessing them. These treatments help avoid the repeatedly counting of evidences. Besides, to avoid the imprecision caused by a rough type-based call graph, we restrict the upper bound of the analyzed call depth to 3 when collecting the fields accessed by a method in computing CoOccur.

The current implementation has not considered the effects of unnested synchronizations. If an object in an inner synchronization can be aliased with the lock object of an outer synchronization, we just safely identify the object’s accessed fields as data that could possibly be protected by the outer synchronization. In the implementation, we also have not considered the indirectly wait()/notify() calls in a critical section, since it is hard to precisely locate them on a type-based call graph. Only direct wait()/notify() calls are handled during refactoring.

The use of the tool is almost straight-forward. If a user wants to do synchronization refactoring, he just needs to click a menu item or a toolbar button. The currently focused project in the workspace will be selected for analysis in the background without needing to specify any program entry. After analysis, all the found refactoring candidates will be loaded into a list view and recommended to the user. The user can preview each recommended refactoring to judge whether it is reasonable. Once confirmed by user, the refactoring transformation can be conducted automatically by the tool.

In the paper, we use 5 open source programs as the experimental subjects. As shown in TABLE I, the subjects include JDK, Hsqldb, JGroups, fop, and lucene, which cover several different types of applications. Columns classes, syncs, and KLOC show the number of outmost classes, the number of synchronized blocks or methods, and the number of source code lines, respectively.

A. Results for Split Lock and Convert to Atomic Refactoring

Our refactoring tool finds Split Lock refactoring candidates in JDK, Hsqldb, and JGroup. The results are listed in TABLE II. They are obtained with the synchronization strength threshold set to 0.3 under a Windows 7 laptop with Intel i7-3720QM CPU and 16G memory. In the table, column class lists the classes whose fields can be protected by split locks, column field partitions lists the partitions of fields each corresponding to a new lock. We have manually checked each reported refactoring candidate. According to our understanding, we believe all of these refactorings are behavior-preserving. For the subjects fop and lucene, since there are only a few critical sections in the source code and the programs are carefully optimized, we did not find any refactoring opportunities in them.

For those found Split Lock refactoring candidates, take Hsqldb as an example. The tool recommends class org.hsqldb.test.JDBCBench as a refactoring target (see Fig. 5). In this class,
fields \textit{transaction\_count} and \textit{failed\_transactions} are suggested to be protected by separate locks. By manual analysis, we found that field \textit{transaction\_count} is used to record the total number of transactions and field \textit{failed\_transactions} is used to record the number of failed transactions. They are never accessed together in the same critical sections. The lock protections on them are used to ensure atomicity of the increment operations. From the structure of the program, we can see that there is no need to keep these two fields consistent via synchronizations. Therefore, this Split Lock refactoring recommended by the tool is reasonable. Fig. 5(b) shows the code after refactoring.

In Fig. 5(b), fields \textit{failed\_transactions} and \textit{transaction\_count} are now uniquely protected by their corresponding locks and the operations on them match to method \texttt{getAndIncrement()} of atomic type \texttt{AtomicInteger}. Obviously, the code satisfies the conditions of Convert to Atomic refactoring. Therefore, these two fields will be identified as Convert to Atomic refactoring candidates. They can further be replaced by \texttt{AtomicIntegers}. Fig. 5(c) shows the code after Convert to Atomic refactoring.

Besides subject Hsqldb, for the candidate classes listed in TABLE II, we also found several other Convert to Atomic refactoring opportunities in the code after split lock refactoring, such as field \texttt{checkId} of class \texttt{com.sun.jmx.snmp.internal.SnmpEngineImpl} in JDK and the fields of class \texttt{org.jgroups.protocols.FRAG2} in JGroups. These refactorings may also help the evolution of concurrent code.

In addition to 0.3, we also tested other threshold values. The found synchronization requirements vary; however, the split lock refactoring candidates do not change. The reason is because, for two fields, whatever there are synchronization requirements between them, if they are once protected in the same critical sections, we cannot assign different locks for them during split lock refactoring (see IV.A). Such restriction limits the number of refactoring candidates. To find more refactoring opportunities, the identification precision of the relationships between locks and their protected data still need to be improved.

B. Results for Split Critical Section Refactoring

We do Split Critical Section refactoring also firstly with the synchronization strength threshold set to 0.3. For this kind of refactoring, the analysis time is very close to that of the Split Lock refactoring. In the 5 experimental subjects, we found three refactoring opportunities which seem to be reasonable: method \texttt{newSystemId()} in class \texttt{com.sun.corba.se.impl.ora.pos.POAPolicyMediatorBase} of JDK (split before line 124), method \texttt{getComponentsByNamespace(short, String)} in class \texttt{com.org.apache.xerces.internal.impl.xx.XSModelImpl} of JDK (split before line 305), and method \texttt{run()} in class \texttt{org.apache.lucene.demo.html.ParserThread} of subject lucene (split before line 40).

Taking the candidate in lucene for example, the splittable critical section in method \texttt{run()} protects two instance fields \textit{summary} and \textit{titleComplete}. The former holds some summary information, while the later indicates whether the parsing of title have been finished. These two fields are used in two threads: a main thread and a working thread used for parsing. The main thread contains the main program logic, while the working thread just does the time-consuming parsing work. Fig. 6 shows the basic structure of these two threads. In this code, only the critical section in method \texttt{run()} protects both \textit{summary} and \textit{titleComplete}. No other place accesses them together in a critical section to ensure their consistency, even when reading information in the main thread. The synchronization strength between \textit{summary} and \textit{titleComplete} is weak. We believe this situation indicates that there is no need to use synchronization to keep the two fields consistent. Therefore, the critical section in method \texttt{run()} can be split into two smaller ones. We manually inspected the transformed code.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig6.png}
\caption{Refactorable synchronization structure in lucene}
\end{figure}
and found the refactoring is behavior-preserving. This refactoring is small, but illustrates the idea of our work.

Including those reasonable candidates, we totally found 48 refactoring candidates in all the experimental subjects: JDK (43), lucene (1), fop (1), Hsqldb (0), and JGroups (3). The number of the reasonable ones seems to be small in all those reports. To seek for more valuable refactoring candidates, we increase the threshold of synchronization strength to 0.5, so that fewer field pairs are considered having synchronization requirements. In this case, the number of all reports is doubled. However, no more reasonable refactoring candidate is found. We consider the reason of this phenomenon is because that the Split Critical Section refactoring opportunities in real public projects are few since the data shared by multiple threads are few and most of the critical sections are carefully written. To reduce the number of false reports, we also decrease the threshold of synchronization strength to smaller values. In this case, more synchronization requirements are identified. The number of false reports significantly decreases. However, when setting a threshold smaller than 0.2, some useful reports may also be missed.

C. Discussions

Although effective, the number of refactoring opportunities found by our tool is still small, especially for the Split Critical Section refactoring. The reasons are at least two-fold. (1) We use rough type-based analyses in determining the alias relation between locks and their protected objects, building call graph, and computing side-effects for each method. This is fast and easy to be safe for library code, but may cause the tool missing some refactoring opportunities. (2) We chose some widely known projects as the benchmarks, however, such projects are often well-written and contain limited number of synchronization refactoring opportunities. Actually, we have read a large number of critical sections in these subjects. Most of them are very small and just access a few key data. It is difficult to further optimize them at source code level, especially to split them. In the future, we plan to find some open and relatively immature subject programs to better illustrate the effectiveness of the proposed approach.

As discussed before, the tool may also generate false reports. There could be several possible reasons. First, our analysis is based on an assumption on the synchronization structures, while the assumption is not always true. Second, the precision of the tool is still limited, and we have ignored some indirect method calls during analysis. Third, the synchronization requirements are inferred according to some experience instead of fully based on program semantics. In the experiment, the Split Lock refactoring suffers less from the false report problem because the fields protected under a previous critical section are still protected together in the new critical section, although with the corresponding lock split and replaced. We do not break their connections as much as Split Critical Section refactoring. To overcome the false report problem, in the future, we plan to weaken the research assumption and design more precise algorithms for refactoring candidate search. Besides, we also need to find better rules to more safely determine the synchronization requirements.

D. Threats to Validity

There are several factors which may threaten the validity of the experimental study. First, since different subject programs have different characteristics, the experimental conclusions may not be able to extend to other projects. Second, although carefully tested, the tool that we implemented may still contain bugs and the experimental data may suffer from such potential bugs. For this, we will continuously check the tool to ensure its correctness. Third, we manually inspected each reported refactoring candidate to determine whether it is reasonable. Since the subject programs are large and complex, there can be mistakes in our judgments. However, the reports have been checked for several rounds. We believe after carefully checking, mistakes in the results are rare.

VI. RELATED WORK

In the past, people have suggested use compilers to optimize the synchronizations [2-6, 16, 17]. Among these researches, Diniz and Rinard [16, 17] coarsen overly frequent synchronizations on the same lock to reduce lock overheads. Aldrich [2], Stoodley and Sundaresan [6] optimize nested and repeated synchronizations on the same lock to avoid redundant lock operations. Some other approaches use escape analysis to eliminate unnecessary synchronizations on non-shared locks [2-5]. Different from the above work, we use source-code level refactoring to refine synchronizations. Compared to bytecode level optimization, source code level refactoring has continuous effects for the whole lifetime of software. Besides, we split locks and split critical sections to refine the synchronization granularity, which is never considered in the existing work on compiler optimizations.

For the refactoring of concurrent programs, the approach proposed by Greenhouse et al. [8] and LockSmith IntelliJ IDEA plug-in [11] support Split Lock, Split Critical Section, and Convert to Atomic refactorings. However, as discussed in the introduction section, they require lots of manual efforts and are not as automated as our approach. Besides them, Lea et al. [9] propose Split Classes, Isolate Field, and other refactorings to refine the synchronization granularity. Goetz [7] propose Lock Striping refactoring to optimize synchronizations on graphs and such like structures. Greenhouse et al. [8] summarize several refactorings including Synchronize Method, Synchronize Callsite, and etc. We also present an automated Shrink Critical Section refactoring approach based on dependence and escape analysis in [18]. In addition to the synchronization related refactorings, Dig et al. [10] propose several refactorings for the parallelization of sequential code. Markstrum et al. [19] present an Extract Concurrent parallelization refactoring to parallelize X10 programs. Wloka et al. [20] propose a refactoring approach to make single-threaded programs reentrant. Damevski et al. [21] propose a refactoring technique, called Extract Kernel, to transform a loop written in C into a parallel function that uses NVIDIA’s CUDA framework. Brown et al. [22] propose a language-independent refactoring approach that helps introduce and tune parallelism through high-level design patterns. These approaches also focus on the evolution of concurrent programs, but their research goals are quite different from ours.
The paper only considers the synchronizations created by keyword `synchronized`. Besides this keyword, Java also provides additional flexible locking constructs for synchronization in the `java.util.concurrent` library. Under certain conditions, the use of such constructs can improve performance significantly. Schäfer et al. propose a refactoring approach for transforming the `synchronized` blocks toward flexible locking [12]. However, how to continuously refactor the flexible locking code is still an interesting problem demanding more studies.

In the paper, we infer the synchronization requirements by analyzing the existing code structures. Similar ideas have also been used in other areas. Lu et al. [23] scan the near reads and writes to infer correlated variable accesses which need to be protected by synchronizations, and thereby use the inferred results to detect concurrency bugs. By inferring the correlations, their approach successfully finds many new bugs. A difference between our work and theirs is that we infer synchronization requirements according to the occurrences of class fields in the existing synchronized code blocks while they do inferring by measuring the distance of variable accesses. We assume the program is correct and all the necessary synchronizations have already been coded in the program, while they do not. Their approach is suitable for bug detection, but it is not clear whether the approach is directly applicable for refactoring purpose.

VII. CONCLUSIONS AND FUTURE WORK

This paper proposes an automated refactoring approach for Java concurrent programs based on synchronization requirement analysis. The approach can identify coarse-grained synchronizations in source code and suggest `Split Lock`, `Split Critical Section`, and `Convert to Atomic refactoring` to refine them. Our experiment on several open source projects indicates that the proposed approach does find effective refactoring opportunities and it can transform the code into better forms automatically.

Although effective, there are still some limitations in our work. First, the precision of our tool still needs more improvement in order to find more refactoring opportunities and reduce false reports. Second, our approach depends on a research assumption and its safety is not fully guaranteed. Third, the approach currently cannot support the analysis of static fields. We plan to address these problems in the future and conduct more experiments to further validate the effectiveness of the proposed approach. Besides, we also plan to automate other concurrency related refactorings to better support the evolution of concurrent software.

REFERENCES