Refactoring Java Programs using Concurrent Libraries

Kazuaki Ishizaki
IBM Research – Tokyo

Shahrokh Daijavad
IBM Research – T. J. Watson Research Center

Toshio Nakatani
IBM Research – Tokyo

kiszk@acm.org

ABSTRACT

Multithread programming is becoming ever-more important to exploit the capabilities of multicore processors. Versions of Java prior to version 5 provide only the synchronized construct as a consistency primitive, which causes a performance scalability problem for multicore machines. Therefore, Java 5 added the java.util.concurrent package to reduce lock contention. Programmers must manually rewrite their existing code to use this package in existing programs. There are two typical rewritings methods. One is to replace an operation on a variable within a synchronized block with an atomic-lock-free version. The other is to replace a sequential concurrent class with its concurrent version. The conventional rewriting approach has three deficiencies. One problem is transformations that may change the behavior of a program. The second problem is missed modifications to be rewritten. The third problem is two difference writing techniques are applied individually to each code fragment even in the same method. This paper describes our refactoring algorithms that address these three problems as they rewrite Java code for scalable performance. We use inter-procedural pointer analysis and consistency tests among the candidate code fragments.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Performance

Keywords

Refactoring, Java, concurrent library, pointer analysis;

1. INTRODUCTION

Multithread programming is becoming increasingly important in exploiting the capabilities of multicore machines. Java supports multithreading both at the language level and in the Java virtual machine. At the language level, a region that cannot run concurrently is declared with a synchronized construct using an object, which creates a critical section to protect its variables. With the synchronized construct, Java provides many thread-safe classes in the standard Java class library. This allows programmers to easily write thread-safe multithreaded programs.

Versions of Java language prior to version 5 provide only the synchronized construct for developers writing concurrent programs. When lock contention involving the synchronized construct occurs, the scalability is limited. This is because the synchronized construct is implemented with monitor semantics [6] and blocking of the contented locks causes heavy overhead. To address this problem, Java 5 [11] introduced the new java.util.concurrent (j.u.c.) package for writing concurrent programs. The classes in the j.u.c.Atomic package provide thread-safe and lock-free primitives to control individual variables. Its collection classes are optimized for scalable performance by reducing the lock contentions.

To gain this scalable performance, programmers must rewrite existing programs to use new libraries [5]. Manual rewriting is labor intensive because many program fragments must be modified consistently across all of the source files. This work is error-prone, because the programmer cannot easily and consistently modify all of the occurrences of the sequential classes. The rewriting work is also omission-prone, because a programmer may miss opportunities to exploit the concurrent APIs [4]. In particular, if a programmer is not familiar with multicore programming, there may be many such errors and omissions. Therefore, previous work [4, 14, 18] provided refactoring tools for improving the concurrency of Java programs. Based on investigations of real programs, previous work [3, 8] proposed two typical refactoring methods.

One is *atomic refactoring* that allows the programmer to use an operation on a variable with a compare-and-swap operation instead of using the synchronized construct. This replaces operations on variables within synchronized blocks with corresponding atomic versions in the j.u.c.Atomic package. For example, the j.u.c.Atomic.AtomicInteger.addAndGet() method offers high scalability by using an atomic operation implemented as a compare-and-swap operation.

The other is *collection refactoring* that allows the programmer to use a thread-safe and highly scalable implementation of the corresponding collection class instead of the sequential implementation in the Java class library. This replaces sequential collection classes in the java.util (j.u.) package with their concurrent versions.

PADTAD'11, July 17, 2011, Toronto, ON, Canada

Table 1. Transformations for *atomic refactoring*

Before transformation	After transformation

`int sharedInt = 0;`
`synchronized (lockObject) {`
`  sharedInt = sharedInt + 1;`
`}`
	`j.u.c.Atomic.AtomicInteger sharedInt;`
	`sharedInt = new AtomicInteger(0);`
	`sharedInt.addAndGet(1);`

`long sharedLong = 0;`
`synchronized (lockObject) {`
`  sharedLong = sharedLong + 1;`
`}`
	`j.u.c.Atomic.AtomicLong sharedLong;`
	`sharedLong = new AtomicLong(0);`
	`sharedLong.addAndGet(1);`

Table 2. Transformations for *collection refactoring*

Before transformation	After transformation
`new j.u.Hashtable()`	`new j.u.c.ConcurrentHashMap()`
`j.u.Collections.synchronizedMap(` `new j.u.HashMap())`	`new CHashMap()` `  // permits null values and the null key`
`j.u.Collections.synchronizedMap(` `new j.u.WeakHashMap())`	`new CWeakHashMap()`
`j.u.Collections.synchronizedSet(` `new j.u.HashSet())`	`new CHashSet()`

A previous system [4] supported these two refactorings. It could easily implement refactoring and transform most target code fragments. However, three problems remain:

a) This approach may change the behavior of programs. [13] described such a problematic transformation.

b) This approach may miss some modifications to be rewritten.

c) This approach requires a programmer to choose an appropriate refactoring approach for each code fragment even in the same method. This may miss refactoring opportunities if the programmer is unfamiliar with the code fragments to being rewritten for higher concurrency.

This paper presents a refactoring approach that addresses these three problems. To address Problem a), we introduce our consistency tests among code fragments. The consistency tests determine whether a transformation can be applied without changing the semantics. To address Problem b), we devised a new refactoring algorithm for *atomic refactoring* using inter-procedural pointer analysis [1], which is derived from [2]. The inter-procedural analysis can allow us to capture all of the candidates to be rewritten. To address Problem c), we use a refactoring algorithm that accepts an object reference as an input. This allows a programmer to choose an entire method at one time as the candidates for two refactoring methods. Our approach supports both *atomic refactoring* and *collection refactoring*.

We implemented the tool as extensions to the Eclipse Java development tools (JDT) and to the language toolkit (LTK). Here is how our tool works:

1. The programmer identifies a method in program fragments that frequently causes lock contention at runtime. A runtime profiler such as Performance Inspector [7] or Oracle Solaris Studio Performance Tools [12] helps with this identification.

2. The method identified in Step 1 is passed to our tool. Our tool finds the object references for possible transformation candidates by scanning for the method. This addresses Problem c).

3. Our tool suggests candidates of program fragments to be changed for the scalability-improving modifications by using a global analysis and the consistency tests. This addresses Problems a) and b).

4. If the programmer accepts the candidates, the appropriate changes are made to all of the source files.

Our tool supports the two transformations in Table 1 and the four transformations in Table 2. Except for the first transformation in Table 2, the `j.u.c.` package itself does not provide a concurrent version. Therefore, we provided our own concurrent versions (`CHashMap`, `CWeakHashMap`, and `CHashSet`) for the other transformations. The `CHashMap` is a hash class that permits null values and the null key while `j.u.c.ConcurrentHashMap` does not accept null values and the null key.

This paper presents the following contributions:

- A new algorithm for *atomic refactoring* to address Problems a) and b) in all of the source files using a pointer analysis and consistency tests (see Section 3.1).
- Application of both transformation approaches in one step to address Problem c) (see Section 3.3).
- An implementation of our refactoring approach as an Eclipse plug-in refactoring tool (see Section 4).

2. MOTIVATING EXAMPLE

This section describes the problems of a conventional refactoring approach using a motivating example. Then it describes how to apply our refactoring approaches that address these problems.

2.1 Problems in a Conventional Refactoring Approach

Figure 1 shows a small Java program used as a running example. The program consists of `Data.java` and `Main.java`. The `Main` class extends the `Data` class. This program first stores an instance of the `j.u.HashMap` class wrapped in the `j.u.Collections.synchronizedMap()` method into a `sharedMap`

variable, and stores one instance into a sharedInt variable. Though the HashMap class is not thread-safe, the returned instantiation is thread-safe, since this wrapping technique delegates the method calls such as HashMap.get() and HashMap.put() using synchronized blocks, as shown in Figure 2. Then the Thread.start() method launches two threads. One executes the GetIncThread.run() method in an infinite loop. The other executes the PutIncThread.run() method in an infinite loop. The loops for both threads access the sharedMap and sharedInt variables at the same time. The sharedMap variable is used as a receiver object to call the delegated SynchronizedMap.get() and SynchronizedMap.put() methods in the j.u.Collections class. The sharedInt variable is updated within a synchronized block. Therefore, these accesses are also thread-safe.

```
 1: // Data.java
 2: import java.util.*;
 3: public class Data {
 4:   protected Object lockObject = new Object();
 5:   static protected Map sharedMap;
 6:   protected int sharedInt = 1;
 7: }
 8:
 9: // Main.java
10: import java.util.*;
11: public class Main extends Data {
12:   public static void main(String args[]) {
13:     Main m = new Main();
14:     sharedMap = Collections. synchronizedMap(new HashMap());
15:     m.sharedInt = m.sharedInt * 2;
16:     new Thread(m.new GetIncThread()).start();   // run GetIncThread
17:     new Thread(m.new PutIncThread()).start();   // run PutIncThread
18:   }
19:
20:   public class GetIncThread implements Runnable {
21:     public void run() {
22:       while (true) {
23:         sharedMap.get(this);
24:         synchronized (lockObject) {
25:           sharedInt = sharedInt + 1;
26:         }
27:       }
28:     }
29:   }
30:
31:   public class PutIncThread implements Runnable {
32:     public void run() {
33:       //synchronized (lockObject) { }
34:       while (true) {
35:         sharedMap.put(this, new Object());
36:         synchronized (lockObject) {
37:           sharedInt = sharedInt - 1;
38:         }
39:       }
40:     }
41:   }
42: }
```

Figure 1. Example program

```
43: public class Collections {
44:   public static <K, V> Map<K, V> synchronizedMap(Map<K, V> map) {
45:     return new SynchronizedMap<K, V>(map);
46:   }
47:
48:   static class SynchronizedMap<K, V> implements Map<K, V> {
49:     private final Map<K, V> m;
50:     final Object mutex;
51:     SynchronizedMap(Map<K, V> map) {
52:       m = map;
53:       mutex = this;
54:     }
55:     public V get(Object key) {
56:       synchronized (mutex) { return m.get(key); }
57:     }
58:     public V put(K key, V value) {
59:       synchronized (mutex) { return m.put(key, value); }
60:     }
        ...
61:   }
     ...
62: }
```

Figure 2. Source code of the java.util.Collections class

This program has two performance problems that limit performance scalability on a multicore machine.

1. Lock contentions will occur frequently at the synchronized blocks in Lines 24 and 36 in Figure 1.
2. Lock contentions will occur frequently at the synchronized blocks in the SynchronizedMap.get() and SynchronizedMap.put() methods.

To address these performance problems, we would like to apply these two transformations to the target program in Figure 1:

1. Use the AtomicInteger.addAndGet() method instead of an operation on the sharedInt variable within a synchronized block (*atomic refactoring*)
2. Use the concurrent version to call new CHashMap() instead of the original sequential collection class synchronizedMap(new HashMap()) (*collection refactoring*)

First, we applied these two refactorings using the conventional refactoring techniques [4] with *CONCURRENCER* 1.0.0. Comparing to Figure 3, which is the refactored program that we expected, we confirmed these four problems:

1. The statement in Line 16 of Figure 3 was not transformed.
2. The statement in Line 17 of Figure 3 was not transformed.
3. *Atomic* and *collection refactorings* were not applied by one action even when there were candidates even within the same method. For example, in Figure 1, Line 24-26 should be selected and applied *atomic refactorings* by a programmer, and then Line 23 should be selected and applied *collection refactorings* independently.
4. If the statement in Line 33 of Figure 1 is enabled (where it is currently commented out) and the refactoring for AtomicInteger can be attempted, this refactoring should not rewrite any code fragment in that case. This is because the synchronized block using the lockObject in Line 33 does not include any operation on the sharedInt variable and the refactored code would not synchronize this statement with the sharedInt.addAndGet() method [13].

Problem b) causes 1 and 2 since the conventional approach cannot detect these two patterns. One reason is that an object that is instantiated elsewhere in a variable declaration for *collection refactoring*. The other reason is a variable that is referenced outside of a synchronized block for *atomic refactoring*. Problem c) causes 3, since the programmer must apply each of the refactorings individually. If it is not known which refactoring can be used, the opportunities to use the concurrent APIs will be missed. Problem a) causes 4, since a transformation that changes the semantics may be used unless the entire program is considered.

To avoid these four problems, our approach uses inter-procedural analysis for all of the source files to address 1 and 2, and uses consistency tests among the candidate code fragments to address 4. Both of our refactorings, which will be described in Sections 2.2 and 2.3, accept a given object as the input for each refactoring to address 3. When a method is given, our approach enumerates all of the object references in the method and then tries to apply the refactorings. For example, if a programmer chooses the GetIncThread.run() method in Figure 1, our approach enumerates sharedMap, this, and sharedInt as the set of given objects. Then all three of these references can be tested with our refactorings.

2.2 Outline of Our Atomic Refactoring

First, we describe how to apply our *atomic refactoring*. For the refactoring for AtomicInteger in Figure 1, we use an object reference for a synchronized block. Our *atomic refactoring* consists of these steps:

1. Find all of the synchronized blocks with the same object as a given lock object using inter-procedural analysis for all of the source files.
2. Check that all of the synchronized blocks found in Step 1 are conformed to two rules that (a) all of the blocks must include the same instance variable and (b) all of the blocks have operations on the variable that can be replaced with an atomic version of those operations.
3. Get the object reference to the instance variable found in Step 2.
4. Find all of the classes whose types are the same as or a superclass of a type of the referenced object found in Step 3.
5. Rewrite the declaration of the specific instance variable found in Step 2 in all of the classes found in Step 4. For example, replace "int var = 0" with "AtomicInteger var = new AtomicInteger(0)".
6. Find all of the object references for the classes found in Step 4 based on inter-procedural analysis in all of the source files.
7. Rewrite all of the object references found in Step 6. (a) If a reference is within one of the synchronized blocks found in Step 2, then rewrite a pair of the synchronized block and its operation using AtomicInteger.addAndGet(). (b) If a reference is outside of the synchronized blocks found in Step 2, then rewrite the references on the left-hand side using AtomicInteger.set() and rewrite the references on the right-hand side using AtomicInteger.get().

Figure 1 is an example. We use a lockObject variable at line 24 as an object reference for a synchronized block. The two synchronized blocks between Lines 24 and 26 and Lines 36 and 38 use the same referenced object. This is because the referenced object Object is instantiated in Line 4. The two synchronized blocks includes operations that increment the same sharedInt instance variable. The object reference of the sharedInt variable is a hidden this variable. Its declaring class is the Main class, and its super-class is the Data class. The declaration of the sharedInt variable is found in the Data class at Line 6. Our tool replaces this declaration "int sharedInt = 1" with "AtomicInteger sharedInt = new AtomicInteger(1) ".

A class object for the hidden this variable is instantiated as the Main class at Line 13. The sharedInt variable with the instantiated class is used at Lines 15, 25, and 37. Since the references in

```
 1: // Data.java
 2: import java.util.*;
 3: import java.util.concurrent.Atomic.AtomicInteger;
 4: public class Data {
 5:   protected Object lockObject = new Object();
 6:   static protected Map sharedMap;
 7:   protected AtomicInteger sharedInt = new AtomicInteger(0);
 8: }
 9:
10: // Main.java
11: import java.util.*;
12: import java.util.concurrent.Atomic.AtomicInteger;
13: public class Main extends Data {
14:   public static void main(String args[]) {
15:     Main m = new Main();
16:     sharedMap = new CHashMap();
17:     m.sharedInt.set(m.sharedInt.get() * 2);
18:     new Thread(m.new GetIncThread()).start();   // run GetIncThread
19:     new Thread(m.new PutIncThread()).start();   // run PutIncThread
20:   }
21:
22:   public class GetIncThread implements Runnable {
23:     public void run() {
24:       while (true) {
25:         sharedMap.get(this);
26:         shareInt.addAndGet(1);
27:       }
28:     }
29:   }
30:
31:   public class PutIncThread implements Runnable {
32:     public void run() {
33:       //synchronized (lockObject) { sharedMap.clear(); }
34:       while (true) {
35:         sharedMap.put(this, new Object());
36:         shareInt.addAndGet(-1);
37:       }
38:     }
39:   }
40: }
```

Figure 3. Refactored program

Line 15 are outside of a synchronized block, our tool replaces the references on the left-hand side and on the right-hand side with the sharedInt.set() and sharedInt.get() methods, respectively. Since the references in Lines 25 and 37 are within the synchronized blocks, our tool replaces each pair of the synchronized block and operation with sharedInt.addAndGet(1) and sharedInt.addAndGet(-1), respectively.

2.3 Outline of Our Collection Refactoring

We describe how to apply our collection refactoring. For the refactoring of HashMap in Figure 1, we use an object reference to the Map class or its subclass. Our *collection refactoring* consists of these steps similar to [2]:

1. Transitively find all of the class instantiations that are reachable from a given receiver object of a method invocation using inter-procedural analysis in all of the source files.
2. Find all of the object references in arguments (for which the external libraries cannot be changed) and which are reachable from all of the class instantiations found in Step 1.
3. Check whether the concurrent version of the class instantiations found in Step 1 is a subclass of the types of the object references found in Step 2.
4. Replace all of the class instantiations found in Step 1 with the corresponding concurrent versions.

Figure 3 is an example of our refactoring. We use the sharedMap on Line 23 as an object reference in Step 1. The HashMap class instantiation through the synchronizedMap() method at Line 14 is referred to via the same reference. This instantiation can reach the references in a sharedMap variable at Lines 23 and 34, but cannot reach any arguments for the external libraries. Our tool replaces the class instantiation at Line 14 with "new CHashMap()", which is the concurrent version of the HashMap class. This class is still a subclass of the Map class.

3. ALGORITHM

This section describes algorithms for *atomic refactoring* and *collection refactoring*. Then it describes how to apply these two algorithms as one action of a programmer.

Notation

r : a given object reference with a synchronized block
P : a input program
HeapLocations[] : mappings from an object reference to instantiations
Pointers[] : mappings from an instantiation to object references
iv: instance variable
ivc: a class
I : set of class instantiations
T: set of object references
S: set of synchronized blocks

```
1:  Subroutine refactoringAtomicClass(r, P, HeapLocations[], Pointers[])
2:     (iv, S) = Detect(r, P, HeapLocations, Pointers)
3:     Rewrite(iv, S, P, HeapLocations, Pointers)

4:  Subroutine Detect(r, P, HeapLocations[], Pointers[])
5:     s = getSynchronizedBlock(r)
6:     iv = getInstanceVariablesInSynchronizedBlock(s)
7:     if |iv| ≠ 1                                   // give up due to more than one instance variable in a synchronized block
8:        return {}
9:     S = {}
10:    I = {}
11:    Worklist = { r }
12:    repeat until fixpoint (i.e. I is not changed)  // find all of the synchronized blocks using the object reachable from r transitively
13:       T = {}
14:       foreach r ∈ Worklist
15:          S = S ∪ getSynchronizedBlock(r)          // get a synchronized block having the lock object r
16:          I = I ∪ HeapLocation[r]
17:          foreach i ∈ HeapLocation[r]
18:             T = T ∪ Pointers[i]
19:       Worklist = T
20:    if |I| ≠ 1                                    // give up due to more than one instantiation for the lock object r
21:       return {}
22:    foreach s ∈ S
23:       if iv ≠ getInstanceVariablesInSynchronizedBlock(s)   // give up due to different instance variable in the synchronized block
24:          return {}
25:       if AtomicMappingCheck(P, s, iv) = ϕ         // give up due to no replaceable atomic operation in the synchronized block
26:          return {}
27:    return (iv, S)

28: Subroutine Rewrite(iv, S, P, HeapLocations[], Pointers[])
29:    ivc = getDeclaredClass(iv)
30:    foreach c ∈ Classes(P)                         // rewrite all of the variable declarations for iv
31:       if c = ivc || c is superclass of ivc
32:          rewrite declaration and initialization for iv in c using AtomicInt
33:    foreach i ∈ getInstanceVariables(P)            // rewrite all of instance variable accesses in P
34:       if iv = i
35:          foreach s ∈ S
36:             if i is within s                      // replace an operation within a synchronized block using Table 3.
37:                AtomicMappingReplace(P, s, i)
38:             else
39:                if i is on LHS                     // replace an reference of iv outside of any synchronized block
40:                   Replace iv with iv.set()
41:                else
42:                   Replace iv with iv.get()
```

Figure 4. The algorithm for *atomic refactoring*

Table 3. Mapping patterns in AtomicMappingReplace(P, s, i)
(e denotes an expression, x denotes a local variable)

Synchronized block s and instance variable i	Replaced pattern

`synchronized (lockObject) {`
`  i = i + e;`
`}`
	`i.addAndGet(e);`

`synchronized (lockObject) {`
`  x = i++;`
`}`
	`x = i.getAndAdd(1);`

`synchronized (lockObject) {`
`  x = i--;`
`}`
	`x = i.getAndAdd(-1);`

`synchronized (lockObject) {`
`  x = i;`
`}`
	`x = i.get();`

`synchronized (lockObject) {`
`  i = e;`
`}`
	`i.set(e);`

3.1 Algorithm of Atomic Refactoring

This section describes the algorithm outlined in Section 2.2. Figure 4 shows our pseudocode for *atomic refactoring*. The refactoringAtomicClass() subroutine takes several arguments. The r represents an object reference with a synchronized block in the program P from a client module. The HeapLocations[] and Pointers[] are based on the results of the pointer analysis. The HeapLocations[] represents an array of mappings from an abstract reference to abstract heap locations. In Figure 1, HeapLocations[] includes a mapping {this at Line 24 ==> new Main() at Line 13}. In reverse, the Pointers[] represents an array of mappings from the abstract heap locations to abstract references. For Figure 1, Pointers[] includes a mapping {new Main() at Line 13 ==> [m at Line 15, m at Line 16, m at Line 17, this at Line 23, this at Line 24, this at Line 25, this at Line 35, this at Line 36, this at Line 37]}.

Here we assume that the lockObject variable in Line 4 is given to the object reference r as the input of this algorithm. In Figure 4, Lines 4-27 find the synchronized blocks to be rewritten. Lines 5-8 get an instance variable within a synchronized block using the object reference r. In Figure 1, iv has the sharedInt variable. For Step 1 described in Section 2.2, Lines 11-19 transitively collect all of synchronized blocks that use the same object as the referenced object of r into S and collect all of the instantiations that are reachable to the object reference r into I using the mappings in HeapLocations[] and Pointers[].

For Step 2, Lines 7-8 and 20-26 check the preconditions that must hold to ensure semantics-preserving refactoring. In Lines 7-8, if the number of instance variables within a synchronized block is more than one, we stop this refactoring since the AtomicInteger package does not support an atomic operation on multiple variables. In Lines 20-21, if the number of instantiations for the lock object is more than one, we stop this refactoring since the refactored code cannot not synchronize among synchronized blocks using different instances of the lock object. Lines 23-24 check whether all of the synchronized blocks include the same instance variable, and Lines 25-26 check whether all of the synchronized blocks include replaceable operations for the methods in the AtomicInteger package. The check at Lines 23-24 avoids Problem 4 described in Section 2.1 [13].

Lines 28-42 rewrite the synchronized blocks and the appropriate instance variable. Line 29 gets the declaring class information on the iv. For Figure 1, iv has the hidden this variable and ivc has the Main class. For Steps 3, 4, and 5, Lines 30-32 rewrite the declarations of the iv in ivc and all of the its super-classes. This rewrites the type declaration and the initialization of the variable iv. For Figure 1, the rewrite for the type declaration replaces int with AtomicInteger. The rewrite for the initialization of the sharedInt variable replaces 1 with "new AtomicInteger(1)". For Steps 6 and 7, Lines 33-42 replace the references for the iv in P. If the instance variable i is within one of the synchronized blocks in the S, we replace that pair of the synchronized block and replaceable operation using the AtomicMappingReplace(P, s, i) subroutine. The AtomicMappingReplace() subroutine rewrites the synchronized block s using the mapping from a statement with a synchronized block to a method in the AtomicInteger package. For this example, the mappings are as represented in Table 3. If the instance variable i is outside of any synchronized block in S, we replace the instance variable i using the this on the left-hand side and this on the right-hand side with sharedInt.set() and sharedInt.get(), respectively.

3.2 Algorithm of Collection Refactoring

This section describes the algorithm outlined in Section 2.3. Figure 5 shows our pseudocode for *collection refactoring*. The refactoringCollectionClass() subroutine takes several arguments. The r represents an object reference to a j.u.Map or j.u.Set class in the program P given by a client module. The CollectionMapping[] represents an array of mappings from a sequential collection class to a concurrent collection class. For our example, Table 2 represents the mappings. The HeapLocations[] and Pointers[] are based on the results of the pointer analysis. The pointer analysis calculates which object references can point to which heap locations. The HeapLocations[] represents an array of mappings from an abstract reference to abstract heap locations. Each abstract reference consists of an object reference and a location in a program. Each abstract heap location consists of a class instantiation and a location in a program. For Figure 1, HeapLocations[] includes the mapping {sharedMap at Line 23 ==> new HashMap() at Line 14}. This means that the object referenced at Line 23 is an instance of the HashMap class instantiated at Line 14. Note that we would get the mapping {sharedMap at Line 23 ==> new SynchronizedMap() at Line 45} for the sharedMap variable based on the naïve result of the pointer analysis. We must take special care with an instantiation wrapped by the j.u.Collections.SynchronizedMap class to get its instantiation of a collection class in an argument. In reverse, the Pointers[] represents an array of mappings from an abstract heap location to abstract references. For Figure 1, HeapLocations[] includes the mapping {new HashMap() at Line 14 ==> [sharedMap at Line 14, shared-Map at Line 23, sharedMap at Line 35]}. This means that the HashMap class instantiated at Line 14 is used at Lines 14, 23, and 35.

Here, we assume that the sharedMap variable in Line 23 is passed to the object reference r as the input of this algorithm. In Figure 5, Lines 8-15 calculate a set of instantiations to be replaced with their concurrent versions. For Step 1 described in Section 2.3, Lines 8-15 transitively collect all of the class instantiations and actual arguments that are reachable from the object reference r into I and A using the mappings in HeapLocations[] and Pointers[]. The I includes the set of instantiations to be replaced.

Lines 16-22 check the preconditions that must hold to ensure semantics-preserving refactoring. For Step 3, Lines 16-18 checks whether the I includes a collection class whose concurrent version is not supported. We stop this refactoring if the I includes an unsupported collection class. For Step 2, Lines 19-22 check whether the concurrent version of the class instantiations I is a subclass of the types of the parameters, for which external libraries that cannot be changed and for which the actual argument A is passed to. For example, in the following program fragment, our tool cannot change the source files in the javax.swing.JTree class because it belongs to external libraries. If our tool replaces "new

```
Notation
    r : a given object reference
    P : a input program
    HeapLocations[] : mappings from an object reference to instantiations
    Pointers[] : mappings from an instantiation to object references
    CollectionMapping[] : mappings from a sequential collection class
       to the corresponding concurrent collection class
    I : set of class instantiations
    A, R: set of object references

1:  Subroutine refactoringCollectionClass(
                r, P, CollectionMapping[], HeapLocations[], Pointers[])
2:    I = Detect(r, P, CollectionMapping, HeapLocations, Pointers)
3:    rewrite(P, I, CollectionMapping)        // replace classinstantiations in I
                                              // based on CollectionMapping

4:  Subroutine Detect(r, P, CollectionMapping[], HeapLocations[], Pointers[])
5:    I = {}
6:    A = {}
7:    Worklist = {r}
8:    repeat until fixpoint (i.e. I is not changed)        // find all of the class instantiations and arguments reachable from r transitively
9:      R = {}
10:     foreach r ∈ Worklist
11:       I = I ∪ HeapLocation[r]
12:       foreach i ∈ I
13:         R = R ∪ Pointers[i]
14:     Worklist = R
15:     A = A ∪ { r | r ∈ R : r is used as an actual argument of external libraries}
16:   foreach i ∈ I                                        // check whether all of the classes have their concurrent version or
17:     if CollectionMapping[class(i)] = ϕ                 //    not give up due to no concurrent collection
18:       return {}
19:   foreach a ∈ A                                        // check whether type correctness will be broken or not
20:     foreach i ∈ I
21:       if i is not subclass of class(dummyArgumentClass(a))
22:         return {}                                      // give up due to breaking type correctness
23:   return I
```

Figure 5. The algorithm for *collection refactoring*

Hashtable()" with "new ConcurrentHashMap()", the type correctness would be violated. This is because this replacement would cause a type mismatch between an actual argument (ConcurrentHashMap) and a dummy argument (Hashtable).

```
j.u.Hashtable hash = new j.u.Hashtable();
...
// javax.swing.JTree(j.u.Hashtable h)
Component c = new javax.swing.JTree(hash);
```

For Step 4, the subroutine rewrite() in Line 3 rewrites all of the instantiations in I using the mappings in CollectionMapping[].

3.3 How to Apply Two Refactorings in One Action

The algorithms described in Sections 3.1 and 3.2 accept an object reference r as an input. When a programmer chooses a method for these two refactorings as an input, our refactoring walks the method and passes all of the object references in the method to the two algorithms. Then each of the two algorithms independently determines whether or not its transformation can be applied to each reference. The programmer needs not choose each code segment individually for each of *atomic refactoring* and *collection refactoring* as a refactoring candidate, unlike the conventional approach.

4. IMPLEMENTATION

Our tool is implemented as an extension to the Eclipse Java development tools (JDT) and language toolkit (LTK). This refactoring is implemented as "Transform for reducing lock contention" in the Refactor menu. While a programmer selects a method in a file, this refactoring rewrites all of the related files in an entire Java project.

Our tool accepts a certain method as its input from the programmer where we assumed that an object reference is given for each refactoring in Sections 2 and 3. This allows the programmer to choose an input for the refactoring more easily than by choosing a particular object reference. It also allows our tool to perform several transformations at the same time, since our refactorings complement each other. When a method is given, our tool scans all of the object references in the method, and chooses the references that are used as lock objects with a synchronized construct and are in the j.u.Map or j.u.Set class as inputs for our refactoring.

Our tool uses the pointer analysis performed by the WALA framework [17]. The current implementation uses a context-sensitive variant of Andersen's analysis [1]. Most of methods are analyzed with one level of object sensitivity [10]. To analyze the methods for the collection classes more precisely, Java collection classes with unlimited-depth object-sensitivity are analyzed. As a result, the Java collections from different allocation sites are fully

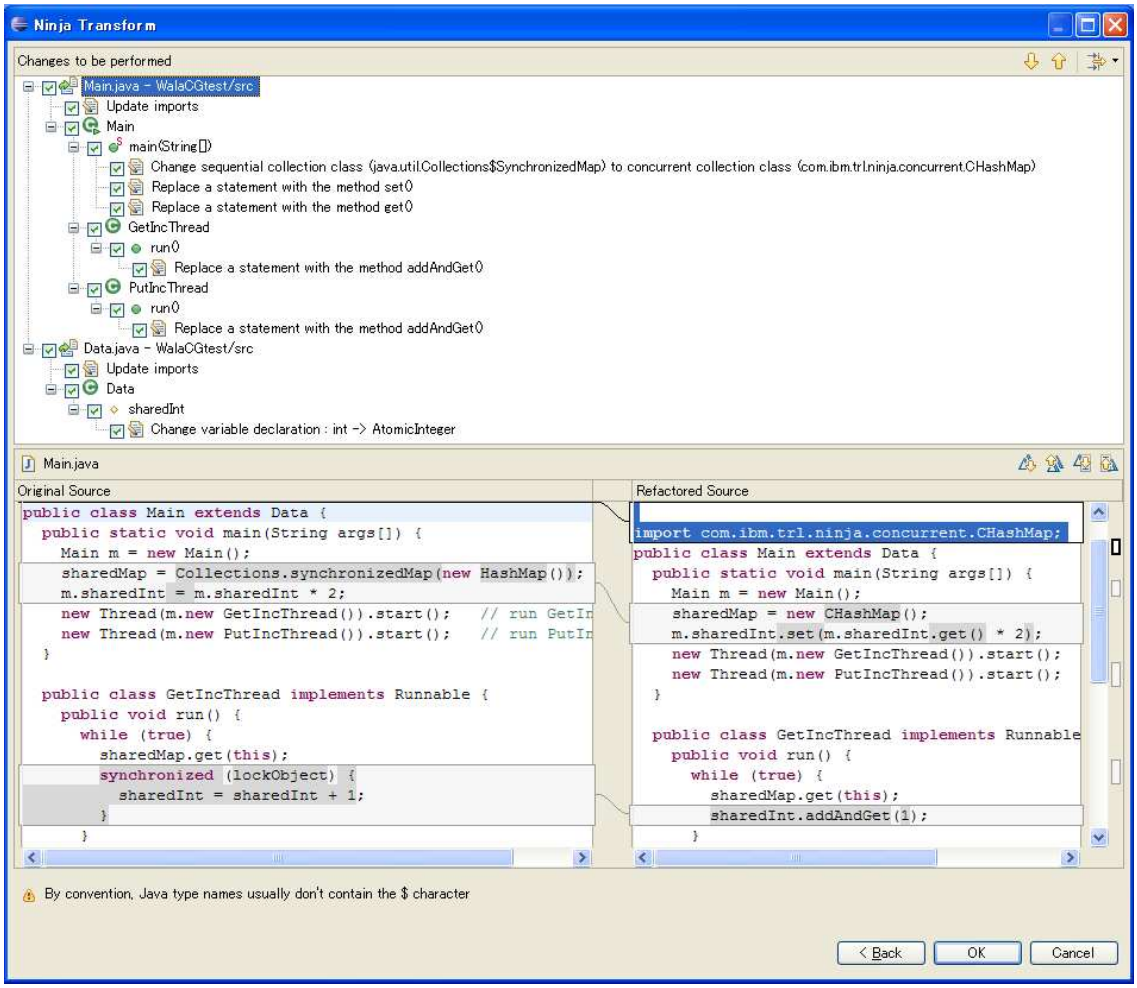

Figure 6. Our tool refactoring to the program in Figure 1

disambiguated so that the results for our refactoring are more precise. For example, since the j.u.Collections.synchronizedMap() method may be called from various points in a program, this handling helps to precisely identify each instantiation of the collection classes from these points.

Figure 6 shows a screenshot as our tool refactors the program of Figure 1. It shows that the refactored source is the same as in Figure 3 and that the *collection refactoring* and *atomic refactoring* are performed at the same time.

5. RELATED WORK

There are several previous papers that present refactoring tools to exploit the concurrency in existing programs.

Dig et al. [4] presented a refactoring tool for Java. This tool, called *CONCURRENCER*, supports three categories of refactoring that include two of our refactoring categories. *CONCURRENCER* replaces a set of related but dispersed API calls with one new API such as the j.u.c.ConcurrentHashMap.putIfAbsent() method. Our tool transforms more refactoring cases for the j.u.c.Atomic.AtomicInteger class, which the current *CONCURRENCER* cannot transform, by using the results of the pointer analysis.

Schäfer et al. [14] presented a refactoring tool for Java. This tool, called ReLocker, supports the use of j.u.c.ReadWriteLock instead of synchronized blocks.

Balaban et al. [2] presented a refactoring tool for Java. This tool transforms usages of legacy library classes for their collection classes to new library classes. A programmer provides the mappings between the old and new APIs. This tool uses point-to information and type-constraint analysis, as also used by our tool. While this tool replaces an API call for collection classes, our tool can also replace an operation within a synchronized block with an API call in the j.u.c.Atomic package.

Wloka et al. [18] presented a mostly automated refactoring tool for Java. This tool makes a program reentrant by replacing global states with thread-local states. While this tool focuses on reentrancy, our tool focuses on shared data accesses. This tool complements our tool.

Markstrum et al. [9] presented a refactoring tool for extracting concurrency from a loop in an X10 program. While this tool fo-

cuses on the scope to be modified within a loop, our tool also supports the program fragments outside of the loop.

Schäfer et al. [13] proposed dependency preservation for synchronization using the synchronization dependence edges to guarantee the correctness of the refactoring of sequential programs. They also pointed out the problem of changing behavior caused by atomic refactoring. We have proposed a solution using interprocedural analysis.

Some work investigated changes to retrofit concurrency into a program. Dig et al. [3] investigated transformations in five open source Java projects. They suggested that it is important to provide hints about what transformations are worth automating. Ishizaki et al. [8] investigated the performance scalability of three commercial Java software products on a multicore machine. They found and addressed performance problems caused by lock contention. They found three categories of solutions to reduce lock contentions. These results led to building a refactoring tool for typical patterns to exploit concurrency in existing programs.

6. CONCLUTION AND FUTURE WORK

We presented our refactoring tool that improves the scalability of existing Java programs for multicore machines. This tool addresses three deficiencies in a conventional approach. One problem is transformations that may change the behavior. The second problem involves missed modifications to be rewritten by a refactoring. The third problem involves different rewritings applied individually target code fragments even in the same method. We use inter-procedural pointer analysis and devised an algorithm to address these three problems in using the `java.util.concurrent` package added in Java 5. We described our algorithms for these two refactoring approaches using pointer analysis. Our tool was implemented in the Eclipse JDT.

We plan to extend our tool to support more refactoring patterns such as double-checked locking [15] as shown in [8]. In addition, we would like to use demand-driven pointer analysis [16] for faster inter-procedural pointer analysis.

ACKNOWLEDGEMENT

We are grateful to the anonymous reviewers for their thoughtful comments and suggestions. We also thank Shannon Jacobs for his editorial assistance.

REFERENCES

[1] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, Denmark, 1994.

[2] Balaban, I., Tip, F., and Fuhrer, R. Refactoring support for class library migration, In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 2005.

[3] Dig, D., Marrero, J., and Ernst, M. D. How do programs become more concurrent? A story of program transformations. Technical Report IT-CSAIL-TR-2008-053, MIT, 2008.

[4] Dig, D., Marrero, J., and Ernst, M. D. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proceedings of international Conference on Software Engineering*, 2009.

[5] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. *Java Concurrency in Practice*, Addison-Wesley, 2006.

[6] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10), 1974.

[7] IBM Corporation. Performance Inspector, http://perfinsp.sourceforge.net/

[8] Ishizaki, K., Daijavad, S., and Nakatani, T. Analyzing and Improving Performance Scalability of Commercial Server Workloads on a Chip Multiprocessor, *IEEE International Symposium on Workload Characterization* (IISWC), 2009.

[9] Markstrum, S. A., Fuhrer, R. M., and Millstein, T. D. Towards concurrency refactoring for x10, *SIGPLAN Note 44(4)*, 2009.

[10] Milanova, A., Rountev, A., and Ryder, B. G. Parameterized Object Sensitivity for Points-to Analysis for Java, *ACM Transactions on Software Engineering and Methodology* (TOSEM), 14(1), 2005.

[11] Oracle Corporation. Java Platform, Standard Edition 5.0, http://java.sun.com/j2se/1.5.0/

[12] Oracle Corporation. The Oracle Solaris Studio Performance Tools, http://download.oracle.com/docs/cd/E18659_01/html/821-2763/gkofq.html

[13] Schäfer, M., Dolby, J., Sridharan, M., Tip, and F., Torlak, E. Correct Refactoring of Concurrent Java Code, In *Proceedings of 24th European Conference on Object-Oriented Programming*, 2010.

[14] Schäfer, M., Sridharan, M., Dolby, J., and Tip, F. Refactoring Java Programs for Flexible Locking, In *Proceedings of international Conference on Software Engineering, 2011*.

[15] Schmidt, D. C. and Harrison, T. Double-Checked Locking – An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently, in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Addison-Wesley, 1997.

[16] Sridharan, M., Gopan, D., Shan, L., and Bodík, R. Demand-driven points-to analysis for Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and applications* (OOPSLA), 2005.

[17] T. J. Watson Libraries for Analysis (WALA), http://wala.sf.net/

[18] Wloka, J., Sridharan, M., and Tip, F. Refactoring for reentrancy, ESEC/FSE '09: *In Proceedings of European Software Engineering Conference and Foundations of Software Engineering Symposium*, 2009.