Refactoring Java Programs for Flexible Locking

Max Schäfer
Oxford University Computing Laboratory
max.schaefer@comlab.ox.ac.uk

Manu Sridharan Julian Dolby Frank Tip
IBM T.J. Watson Research Center
{msridhar,dolby,ftip}@us.ibm.com

ABSTRACT

Recent versions of the Java standard library offer flexible locking constructs that go beyond the language's built-in monitor locks in terms of features, and that can be fine-tuned to suit specific application scenarios. Under certain conditions, the use of these constructs can improve performance significantly, by reducing lock contention. However, the code transformations needed to convert between locking constructs are non-trivial, and great care must be taken to update lock usage throughout the program consistently. We present *Relocker*, an automated tool that assists programmers with refactoring `synchronized` blocks into `ReentrantLocks` and `ReadWriteLocks`, to make exploring the performance tradeoffs among these constructs easier. In experiments on a collection of real-world Java applications, *Relocker* was able to refactor over 80% of built-in monitors into `ReentrantLocks`. Additionally, in most cases the tool could automatically infer the same `ReadWriteLock` usage that programmers had previously introduced manually.

Categories and Subject Descriptors

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

General Terms

Measurement, Performance

Keywords

Refactoring, monitors, read-write locks

1. INTRODUCTION

As multi-core processors are becoming pervasive, programs are becoming more concurrent to take advantage of the available parallelism. However, increasing concurrency in a program is often non-trivial, due to various potential scalability bottlenecks. One common bottleneck is *lock contention*, where scalability is limited by many threads waiting to acquire some common lock in order to safely access shared memory.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Various solutions exist for addressing lock contention, each with benefits and drawbacks. Approaches that avoid locks altogether include lock-free data structures (see, e.g., [17]) and transactional memory (TM) [11]. However, writing correct lock-free data structures requires more expertise than can be expected from most programmers, and the semantics of TM may not be suitable in some cases (e.g, if I/O needs to be performed). Making locking more fine-grained can also increase concurrency, but potentially risks introducing subtle race conditions. In the context of Java, the standard `java.util.concurrent` library (in the sequel abbreviated as `j.u.c`) provides a number of data structures and locking constructs that could also be helpful, with their own tradeoffs. With all these options, there is a strong need for tool support to help programmers experiment with different solutions to see what works best in a particular situation.

In this paper, we focus on refactoring support for the advanced locking constructs available in `j.u.c` [21]. The `ReentrantLock` type enables many features unsupported by Java's built-in locks, such as non-block-structured lock operations, checking if a lock is held (`tryLock()`), interrupting lock acquisition, and specifying fairness behavior under contention. Additionally, the `ReadWriteLock` type enables distinguished reader and writer locks, where multiple threads holding the reader lock may execute concurrently. The goal of our research is to provide refactoring tools that support the transition from built-in locks to these advanced lock types.

Many difficulties arise when manually transforming a program to use the locking constructs of `j.u.c`, motivating better tool support. First, these constructs lack the concise and intuitive syntax of the `synchronized` blocks associated with Java's built-in monitor locks. Instead, locks are modeled as objects, and lock operations as method calls, and the burden is on the programmer to ensure that acquisition and release of locks are properly matched. Second, the relative performance of different lock types strongly depends on the number of threads and their workload, and on the architecture and JVM being used. As we shall show in Section 2, these performance tradeoffs are often unclear, and may change as programs and JVMs evolve. Therefore, programmers may need to switch back and forth between different lock types to determine the best lock for the job. Third, the transformation from one locking construct to another can require tricky non-local reasoning about program behavior. All code blocks using the same lock must be transformed together to ensure behavior preservation, and discovering all such blocks can be non-trivial. In some cases, the migration to advanced locks is impossible when the program extends a framework that relies on a specific form of synchronization. Introducing read-write locks requires careful reasoning about where a read lock is safe to introduce, as incorrect use of a read lock can lead to subtle race conditions.

In this paper, we present *Relocker*, an automated refactoring tool

that can replace built-in monitor locks with ReentrantLocks and ReadWriteLocks.[1] Building a practical tool for performing these lock refactorings is challenging—the transformations involved require knowledge about object aliasing and possible heap side effects, but most analyses for computing such information are not suitable for use in a refactoring tool, due to performance issues or the assumption that all relevant code is reachable from a set of entry points. *Relocker* is carefully designed to enable automation of most of the code transformations needed to switch between locking constructs in real-world programs, while only using analyses suitable for a practical refactoring tool.

The contributions of this paper are:

- Algorithms for converting from built-in monitor locks to ReentrantLocks, and for converting from ReentrantLocks to ReadWriteLocks.
- An implementation of these algorithms in an automated refactoring tool called *Relocker*.
- An evaluation of *Relocker* on a set of Java programs, demonstrating that *Relocker* was able to refactor over 80% of all monitor locks into ReentrantLocks, and showing that, on several programs that already used ReadWriteLocks, *Relocker* was able to infer read locks in most cases where programmers had previously introduced them manually.

The remainder of this paper is organized as follows. Section 2 presents background on the advanced lock types from j.u.c and an example to motivate our refactorings. Sections 3 and 4 present algorithms for converting from monitor locks to ReeentrantLocks, and from the latter to ReadWriteLocks, respectively. In Section 5, we present the implementation of *Relocker* and its evaluation on a set of Java benchmarks. Section 6 discusses related work, and conclusions are presented in Section 7.

2. MOTIVATING EXAMPLE

In this section, we give an overview of our proposed refactorings via three versions of an example class implemented with the different locking constructs. We introduce these constructs and present variants of the example class using each of these constructs in Section 2.1. Then, in Section 2.2, we discuss the complex performance tradeoffs between the variants, motivating the need for a refactoring tool to enable experimentation. Finally, Section 2.3 illustrates some challenges of performing the refactorings in the context of the examples.

2.1 Example

Figure 1 illustrates the different locking constructs involved in our refactorings. The figure shows three different implementations of a class SyncMap, a *synchronization wrapper* similar to the ones available in class java.util.Collections. Each SyncMap implementation handles all map operations by acquiring a lock, delegating to the corresponding operations in a contained Map object, and finally releasing the lock.

The program of Figure 1(a) uses the built-in monitor locks that are associated with Java's **synchronized** blocks. While these locks have the benefit of concise syntax and low overhead in the uncontended case, there are situations where more flexibility is required, and where their performance is suboptimal. To address these shortcomings, two alternative types of locks are available in Java standard libraries since Java 5.0, in package java.util.concurrent.locks:

[1]Refactoring ReentrantLocks and ReadWriteLocks back into built-in monitor locks is easy, provided that none of the features specific to the advanced lock types are used.

JVM	# cores	# readers	# writers	throughput (ops/second) sync'd	reentrant	R/W
Sun 1.6.0_07	8	9	1	528.7	360.9	**1072.3**
Sun 1.6.0_07	8	1	1	**299.6**	206.9	169.4
Sun 1.6.0_07	8	1	9	**273.2**	218.9	181.8
Sun 1.5.0_15	8	9	1	244.3	368.9	**1677.7**
Sun 1.5.0_15	8	1	1	187.7	195.5	**208.8**
Sun 1.5.0_15	8	1	9	**227.4**	179.8	199.2
Sun 1.5.0_17	2	9	1	305.2	288.1	**611.7**
Sun 1.5.0_17	2	1	1	161.6	**176.4**	164.6
Sun 1.5.0_17	2	1	9	**200.9**	175.4	159.5

Table 1: Synthetic benchmark throughput. All measurements are averages of 10 runs of 10 seconds each. 8-core numbers were taken on a 4-way dual-core 3.2GHz Intel Xeon Linux machine with 20GB RAM. 2-core numbers were taken on a 2GHz AMD Athlon 64 X2 Dual Core Linux machine with 4 GB RAM.

- ReentrantLock has similar behavior to a built-in monitor lock, but is more flexible by (i) allowing non-block-structured regions to be protected by locks, (ii) supporting tryLock(), a mechanism for testing whether a lock is held, and (iii) supporting fairness parameters and multiple condition variables.
- A ReadWriteLock has an associated read lock and write lock. Only one "writer" thread may execute while holding the write lock, but multiple "reader" threads can execute concurrently while holding the read lock (as long as no thread holds the write lock). This construct enables better performance when write operations are relatively infrequent.

Both of these types of locks require slightly more awkward syntax than traditional **synchronized** blocks: the programmer creates a lock by calling the constructor for the appropriate lock type, and must call methods on the returned lock object to perform lock()/unlock()/tryLock() operations.

Figure 1(b) shows an alternative implementation of class SyncMap based on ReentrantLocks that is semantically equivalent to the one in Figure 1(a). In this version, the lock is created when the SyncMap-object is being constructed, a call to lock() is inserted at the beginning of each method, and a call to unlock() before returning. Note that a **try-finally** construct must be used in order to ensure that the lock is released when a method exits exceptionally. The solution based on ReadWriteLocks shown in Figure 1(c) is analogous to that of Figure 1(b), but utilizes the read lock, obtained by invoking method readLock(), for methods that do not update the map; only method put() requires the use of the write lock.

2.2 Performance Tradeoffs

The performance differences between the different locking constructs can be both dramatic and unpredictable. Table 1 presents a performance comparison of a synthetic benchmark using our implementations of SyncMap on different JVMs and hardware. The benchmark spawns some number of reader and writer threads that respectively perform random reads (i.e. get and containsKey operations) and writes (i.e. put operations) to a shared SyncMap, and then measures throughput in terms of total operations per second.[2] The final three columns of Table 1 give throughput numbers for the SyncMap implementations using **synchronized** methods, reentrant locks, and read-write locks (Figure 1(a), (b), and (c) respectively).

Clearly, switching to a different type of lock can have significant performance benefits. For example, on the 8-core machine

[2]The full benchmark code is available at http://progtools.comlab.ox.ac.uk/projects/refactoring/relocker.

```
public class SyncMap {
  private final Map map;
  public SyncMap(Map map) { this.map = map; }
  public synchronized Object put(Object k, Object v) {
    return map.put(k, v);
  }
  public synchronized Object get(Object k) {
    return map.get(k);
  }
  public synchronized int size() {
    return map.size();
  }
}
```

(a)

```
public class SyncMap {
  private final ReentrantLock lock;
  private final Map map;
  public SyncMap(Map map){
    this.map = map;
    lock = new ReentrantLock();
  }
  public Object put(Object k, Object v) {
    lock.lock();
    try { return map.put(k, v);
    } finally { lock.unlock(); }
  }
  public Object get(Object k) {
    lock.lock();
    try { return map.get(k);
    } finally { lock.unlock(); }
  }
  public int size() {
    lock.lock();
    try { return map.size();
    } finally { lock.unlock(); }
  }
}
```

(b)

```
public class SyncMap {
  private final ReadWriteLock lock;
  private final Map map;
  public SyncMap(Map map){
    this.map = map;
    lock = new ReentrantReadWriteLock();
  }
  public Object put(Object k, Object v) {
    lock.writeLock().lock();
    try {
      return map.put(k, v);
    } finally { lock.writeLock().unlock(); }
  }
  public Object get(Object k) {
    lock.readLock().lock();
    try {
      return map.get(k);
    } finally { lock.readLock().unlock(); }
  }
  public int size() {
    lock.readLock().lock();
    try {
      return map.size();
    } finally { lock.readLock().unlock(); }
  }
  public int sizeViaIter() {
    lock.readLock().lock();
    try {
      Iterator i = map.entrySet().iterator();
      int count = 0;
      while (i.hasNext()) {
        count++; i.next();
      }
      return count;
    } finally { lock.readLock().unlock(); }
  }
}
```

(c)

Figure 1: Alternative implementations of class `SyncMap` based on standard monitor locks (a), `ReeentrantLock`s (b) and on `ReadWriteLock`s (c).

running the Sun 1.6.0_07 JVM, switching from a monitor-based implementation of `SyncMap` to one based on `ReadWriteLock`s yielded a more than twofold increase in throughput when read operations dominate. When the Sun 1.5.0_15 JVM was used on the same configuration, the version based on `ReadWriteLock`s won by an even more dramatic factor of 5. However, in a configuration where write operations were more prevalent, the version with **`synchronized`** blocks was 50% faster than one based on read-write locks with the Sun 1.6.0_07 JVM (14% faster with the Sun 1.5.0_15 JVM). In a low-contention case with just 1 reader and 1 writer, the performance differences were less extreme, and each of the three types of locks yielded the fastest version on at least one machine/VM configuration (e.g., note that `ReentrantLock`s were fastest on the 2-core machine with the Sun 1.5.0_15 JVM).

As shown by Table 1, the relative performance of different types of locks strongly depends on the mix of read and write operations, and on the architecture and JVM being used. Careful experimentation is needed to determine which locks perform best, and this argues strongly for refactoring tools that make it easy to switch between different types of locks.

2.3 Refactoring Challenges

The *Relocker* tool that we developed is capable of automatically inferring the version of `SyncMap` of Figure 1(b) from the original code in Figure 1(a). While the transformation needed to perform the refactoring is fairly straightforward for the simple example under consideration, a slightly more involved transformation is sometimes needed. Also, determining when the transformation is safe requires non-local reasoning. With the `SyncMap` example, in addition to the code shown, the refactoring must check which other **`synchronized`** blocks in the program might lock a `SyncMap` object. This check requires aliasing information, which is typically computed with expensive whole-program analysis. However, we have devised techniques that are sufficient for handling typical usage of **`synchronized`** blocks and methods while avoiding any costly whole-program analysis. Section 3 details our algorithm for refactoring built-in monitor locks into `ReeentrantLock`s.

Relocker can also automatically refactor from `ReentrantLock`s to `ReadWriteLock`s, transforming the class in Figure 1(b) into that of Figure 1(c). A key goal of this refactoring is to introduce as many read locks as possible, in order to increase potential parallelism in the refactored program. However, determining when a read lock can be used safely can be quite challenging in the presence of heap updates, only some of which are relevant to locking.

Consider the (contrived) `sizeViaIter()` method in Figure 1(c), which computes the size of the map by iterating through its entries. For typical Java `Iterator`s, the `next()` method updates the state of the object to reflect the current traversal position. In this case,

however, this heap update mutates a local object (allocated by the `iterator()` call), and hence does not affect the correctness of using a read lock. Most attempts to prove such program properties in the literature are again based on costly whole-program analysis, but in Section 4 we describe a carefully-designed local analysis that often suffices for the task of read lock inference.

3. INTRODUCING REENTRANT LOCKS

The CONVERT TO REENTRANT LOCK refactoring allows the programmer to replace all uses of a built-in monitor with corresponding uses of an object of type `ReentrantLock`; in particular, it can transform the code from Figure 1(a) to that of Figure 1(b). All operations on built-in monitors have their equivalent on reentrant locks, and moreover the semantics of reentrant locks with respect to ordering, visibility and atomicity is the same as for built-in monitors [21]. Hence this refactoring preserves program behavior as long as it is performed *consistently* in the following sense: two operations on the same built-in monitor in the original program either still operate on the same monitor in the refactored program, or they have both been refactored to work on the same reentrant lock, and vice versa.

Because using `ReentrantLock` objects for locking can carry a higher overhead than using built-in monitors on modern JVMs, this refactoring can degrade program performance. As explained previously, however, reentrant locks offer additional features, such as non-block-structured locking, that may be useful in some circumstances. While this refactoring does not aim to introduce the use of such features into the program, it makes it much easier for the programmer to experiment with them; it also lays the groundwork for a more sophisticated refactoring for converting reentrant locks to read-write locks, introduced in Section 4.

It is, of course, not possible for the refactoring to transform a *single* monitor as it exists at runtime into a reentrant lock: for instance, the same **`synchronized`** instance method will use different monitors when invoked on different objects, and the monitor expression of a **`synchronized`** block may evaluate to different objects at different times. Hence, it is more correct to think of the refactoring as changing a *set* M of monitors into a set L of reentrant locks.

To achieve the goal of consistently refactoring all uses of monitors from M to corresponding uses of locks from L, we have to answer two questions: (1) *which* monitor uses have to be refactored together, and (2) *how* they ought to be refactored. To answer question 1, we must categorize all uses of built-in monitors into those that use a monitor from M (and hence must be refactored) and those that cannot possibly do so. For the second question, we have to uniquely assign a lock from L to every monitor from M, and replace all relevant monitor uses with a use of the corresponding lock.

In principle, the two questions are independent. However, answering the second question leads to a simple and practical answer to the first, so we discuss how to perform the refactoring (Section 3.1) before discussing what code to refactor (Section 3.2).

3.1 How to Refactor

We call a language construct that operates on a built-in monitor a *monitor action*. Every monitor action a has a *monitor expression* $me(a)$ that evaluates to the object whose monitor is accessed by the action. There are four kinds of monitor actions:

1. synchronized instance methods, which enter and exit the monitor of their receiver object; their monitor expression is **`this`**;

2. synchronized static methods, which enter and exit the monitor of the class object for their enclosing class; their monitor expression is a class literal for that class;

```
class C {
  private Map m = new HashMap();
  public Object get(Object k) {
    synchronized(m) { return m.get(k); }
  }
  public void put(Object k, Object v) {
    synchronized(m) { m.put(k, v); }
  }
}
```

Figure 2: Monitor actions on unshared fields

3. synchronized blocks, which enter and exit the monitor of the object their expression evaluates to; that expression is their monitor expression;

4. calls to methods `wait`, `notify` and `notifyAll`, which operate on the monitor of their receiver object; their monitor expression is their receiver argument.

Note that both types of synchronized methods can easily be desugared into synchronized blocks. Hence, we limit our discussion to handling of synchronized blocks and calls to `wait` or `notify`.

For a synchronized block with monitor expression e, let lock expression $l(e)$ evaluate to a corresponding reentrant lock. Given $l(e)$, the block can be rewritten like this:

```
synchronized(e) {
  ...
}
```

$\Longrightarrow$

```
l(e).lock();
try {
  ...
} finally {
  l(e).unlock();
}
```

As the program transformation is straightforward, the main problem in performing the refactoring is determining $l(e)$.

Our strategy for associating a reentrant lock with an object o is to store the lock in a field l of o, which is made public to allow access from any package. This technique is natural, as it matches the association of Java's built-in monitors with unique objects. Syntactically, the refactoring needs to insert a final instance field l of type `ReentrantLock` into the declaration of the class of o, and initialize it to a new instance of `ReentrantLock`. Now, $l(e)$ can simply be defined as $e.l$. This strategy only works if the type of o is a class (since Java interfaces cannot have instance fields) and if the source code of the class is modifiable, i.e., it cannot be a library class.

For static synchronized methods, the instance field insertion strategy does not work: the locked object for a synchronized static method in class C is of type `Class<`C`>`, which cannot be modified. Fortunately, we can achieve the same effect by storing the reentrant lock in a new static field $C.l$ (again, assuming C is modifiable). In this case, we define $l(e)$ to be $C.l$.

This leaves us with the case of monitor actions on expressions whose type is neither a parameterized instance of `Class` nor a modifiable class. Such monitor actions occur quite frequently in real code, and hence cannot be ignored by the refactoring. A common usage pattern of this kind is shown in Figure 2: class `C` has a member field `m`, which is initialized to a fresh object, and is used to synchronize access to other data (in this case the map itself).

We can exploit encapsulation to handle most such cases. Note that the reference stored in `m` is never leaked in any way, so the stored `HashMap` object is only accessible through `m` itself. Consequently, the field satisfies the following important property:

> Any monitor action that operates on the monitor of an object stored in the field must access it through that very field.

We call fields with this property *unshared*.

The field `m` of Figure 2 is unshared, since it is only ever assigned newly created objects, its value is never assigned to another variable, and the methods invoked on it (`HashMap.put` and `HashMap.get`) do not leak the value of their receiver object. In Section 5, we will discuss a simple syntactic check to determine whether a field is unshared.

Suppose we want to refactor the set of monitors associated with all the objects stored in an unshared field like `m`. To associate lock objects with these monitors, we introduce a new lock field `l` into `m`'s enclosing class `C` (which must be modifiable). Every monitor action operating on `m` has a monitor expression of the form e`.m`, so we can easily refactor it into a corresponding lock operation on e`.l`.

To refactor invocations of `wait` and `notify`, we utilize the condition variables of type `Condition` associated with `ReentrantLocks`. Multiple condition variables can be associated with a `ReentrantLock` via calls to `newCondition()`, but for our refactoring only one such variable is needed. The refactoring introduces an additional condition variable field c alongside the lock field l and initializes it to a new variable whenever l is initialized to a new lock. Uses of `wait` and `notify` can then be straightforwardly rewritten into corresponding uses of `await` and `signal` on c.

3.2 What to Refactor

The previous subsection introduced three ways of associating a lock object with a built-in monitor: (1) as an instance field of the type C of the object to which the monitor belongs, (2) as a class field of the type C to whose class object the monitor belongs, and (3) as a sibling field of the unshared field f to whose value the monitor belongs.

This suggests an abstraction of sets of monitors as *abstract monitors* of the following three types:

1. for a type C, the *T-monitor* $\mathrm{TM}(C)$ represents the monitors belonging to all objects of type C or its subtypes;
2. for a type C, the *C-monitor* $\mathrm{CM}(C)$ represents the (single) monitor belonging to the class object of type C;
3. for an unshared field f, the *F-monitor* $\mathrm{FM}(f)$ represents the monitors of all objects stored in f.

For an abstract monitor M, we write $[\![M]\!]$ to denote the set of concrete monitors it represents. We write $M \subseteq M'$ to denote $[\![M]\!] \subseteq [\![M']\!]$, and $M \perp M'$ to denote $[\![M]\!] \cap [\![M']\!] = \emptyset$. For instance, we have $\mathrm{CM}(C) \subseteq \mathrm{TM}$(`java.lang.Object`) for any class C, since `Class<`C`>` extends `java.lang.Object`. Similarly, if field f has type C, then $\mathrm{FM}(f) \subseteq \mathrm{TM}(C)$; and, of course, $\mathrm{TM}(C) \subseteq \mathrm{TM}(B)$ whenever C is a subtype of B.

On the other hand, note that $\mathrm{CM}(C) \perp \mathrm{FM}(f)$ for every C and f: if the class object of C were stored in f, then f would not be unshared, since its value could be accessed as C`.class` without reference to f. Likewise, for two unshared fields f and g, $[\![\mathrm{FM}(f)]\!]$ and $[\![\mathrm{FM}(g)]\!]$ are either equal or disjoint, and they can only be equal if $f = g$.

To determine which monitor actions to refactor, the refactoring assigns to every monitor action a an abstract monitor $M(a)$ that conservatively overapproximates the set of monitors that a could operate on at runtime. Given this assignment, if M is the abstract monitor representing the monitors whose uses are to be replaced with reentrant locks, then all a with $M(a) \subseteq M$ should be refactored, and for all other actions a' we must have $M(a') \perp M$.

A straightforward definition of $M(a)$ is as follows:

$$M(a) := \begin{cases} \mathrm{FM}(f) & \text{if } me(a) \text{ is an access to unshared field } f \\ \mathrm{CM}(C) & \text{if } me(a) \text{ is of type } \texttt{Class<}C\texttt{>} \\ \mathrm{TM}(C) & \text{otherwise, where } me(a) \text{ is of type } \texttt{C} \end{cases}$$

However, this definition does not capture all the information we need. If $M(a)$ is $\mathrm{TM}(C)$, we only know that a operates on the monitor of some object assignable to type C. In fact, however, a also cannot operate on the monitor belonging to any object stored in an unshared field (even if that field is of type C), for otherwise $M(a)$ would have to be an F-monitor.

In order to track this additional information, we slightly modify our definition of $\mathrm{TM}(C)$:

1'. for a type C, the abstract monitor $\mathrm{TM}(C)$ represents the set of all monitors belonging to all objects of type C or its subtypes, *except those stored in unshared fields*.

The definition of $M(a)$ above still gives a sound overapproximation of the set of monitors that a could operate on under this new definition, but it is now very easy to check inclusion and disjointness of abstract monitors.

To describe how we compute inclusion and disjointness, let us first define the *type* $tp(M)$ of an abstract monitor M by stipulating that $tp(\mathrm{CM}(C)) :=$ `Class<`C`>` and $tp(\mathrm{TM}(C)) := C$, whereas $tp(\mathrm{FM}(f))$ is undefined. Then, it is easy to see that

- $M \subseteq M'$ iff either $M = M'$, or $tp(M)$ is a subtype of $tp(M')$;
- $M \perp M'$ iff one of the following holds:
  - M is $\mathrm{FM}(f)$, M' is $\mathrm{FM}(f')$, and $f \neq f'$;
  - M is $\mathrm{CM}(C)$, M' is $\mathrm{CM}(C')$, and $C \neq C'$;
  - $tp(M)$ and $tp(M')$ have no common (reflexive, transitive) subtype.

3.3 The Algorithm

We now describe the refactoring algorithm in more detail. Figure 3 gives a pseudocode description of the main procedure of the refactoring, CONVERT TO REENTRANT LOCK. Given an abstract monitor M to refactor, it creates a corresponding lock field using procedure `createLockField`, and then iterates over all monitor actions a in the program. Those actions that must acquire the same monitor ($M(a) \subseteq M$) are refactored using procedure `transformAction`; for all others, the refactoring ensures that their set of monitors is disjoint from M, and aborts if that is not the case, reverting any changes it has already made.

Procedure `createLockField`, shown in the same figure, analyzes the kind of M and creates the lock field in the appropriate type, ensuring that the type is modifiable. We ignore here shallow issues to do with name binding; for instance, the name of the lock field cannot have the same name as a field already declared in the same class.

Procedure `transformAction` syntactically transforms a given monitor action a into a corresponding action on reentrant locks. We use function `mkLockAccess` to compute an expression l that refers to the reentrant lock object. Finally, `transformAction` performs the appropriate syntactic transformation of the program.[3]

[3]Note that in some cases, l is inserted into the program twice. If l is a complicated expression that should not be evaluated twice, or whose value may have been changed by the block, we can first perform an EXTRACT TEMP refactoring to extract its value into a fresh local variable x, and then perform locking and unlocking on x instead.

```
1: procedure CONVERT TO REENTRANT LOCK (AbstractMonitor M):
2: createLockField(M)
3: for all monitor actions a do
4:   if M(a) ⊆ M then
5:     transformAction(a)
6:   else
7:     assert M(a) ⊥ M

8: procedure createLockField (AbstractMonitor M):
9: if M is FM(f) then
10:   assert f is declared in a modifiable type
11:   create lock field l as sibling field of f
12:   for all assignments a to f do
13:     insert assignment l = new ReentrantLock() after a
14: else if M is CM(C) then
15:   assert C is a modifiable type
16:   create static lock field l in C
17: else /* M must be of the form TM(C) */
18:   assert C is a modifiable class
19:   create lock field l in C

20: procedure transformAction (MonitorAction a):
21: assert a is from source code
22: l ←mkLockAccess(a)
23: if a is synchronized(e) { ... } then
24:   replace a with
      l.lock(); try { ... } finally { l.unlock(); }
25: else if a is a synchronized method then
26:   remove synchronized modifier from a
27:   replace body of a with
      l.lock(); try { ... } finally { l.unlock(); }
28: else /* a must be call to wait or notify */
29:   ...

30: function mkLockAccess (MonitorAction a):
31: e ← me(a)
32: if M(a) is FM(f) then /* e is of the form e'.f */
33:   return e'.l
34: else if M(a) is CM(C) then
35:   return C.l
36: else /* M(a) must be of the form TM(C) */
37:   return e.l
```

Figure 3: Refactoring CONVERT TO REENTRANT LOCK

We have elided the code to deal with refactoring of `wait` and `notify` monitor actions. To handle this, we need a procedure `createConditionField` to create a field to hold the condition variable, which is completely analogous to `createLockField`, and procedure `transformAction` must rewrite the method call in question into an appropriate call on the condition variable field, which is created by a function `mkConditionAccess` similar to `mkLockAccess`.

Observe that this refactoring can fail for three reasons: refactoring the abstract monitor would require refactoring a monitor action that comes from compiled code and is hence not modifiable (Line 21); there are ambiguous monitor actions that cannot be refactored consistently (Line 7); the refactoring would need to modify an unmodifiable type (Lines 10, 15, 18).

4. INTRODUCING READ-WRITE LOCKS

As discussed in Section 2, lock contention can sometimes be reduced through use of a *read-write lock*, which allow threads to read the state protected by the lock concurrently, as long as no other thread is writing the state. In this section, we present a refactoring CONVERT TO READ-WRITE LOCK that enables programmers to more easily experiment with using read-write locks to improve performance. Note that we only describe the refactoring for programs already using reentrant locks; the CONVERT TO REENTRANT LOCK refactoring of Section 3 can be used to introduce such locks if needed.

Our refactoring aims to introduce read locks (rather than write locks) whenever it can prove it is safe to do so, thereby maximizing potential concurrency in the transformed program. Conceptually, a reentrant lock l can only be transformed into a read lock if any code that may execute while l is held does not modify the shared state protected by l. Most Java code does not formally document the relationship between locks and the corresponding protected shared state.[4] Hence, our algorithm checks for *any* modification of potentially shared state while a lock is held, and it only introduces a read lock when no such modifications are found.

[4]When such relationships are documented (e.g., through `GuardedBy` annotations [8]), we can easily use the information in our analysis to potentially infer more read locks.

Figure 4 gives pseudocode for the CONVERT TO READ-WRITE LOCK refactoring. The refactoring takes a field f that must be of type `ReentrantLock`. It changes the type of the field and adjusts any assignments to f, including its initializer if it has one. This step of the refactoring requires that f may only be assigned newly created objects (line 5) in order to maintain a one-to-one correspondence between reentrant locks in the original program and read-write locks in the refactored program.

Now, every use of f is adjusted. To preserve type correctness, all uses of f have to be invocations of its `lock` and `unlock` methods, and we require that these appear in the standard **`try-finally`** pattern seen in earlier examples. In practice, most developers seem to follow this pattern, so this is not a serious restriction. Handling more general cases would require some form of data flow analysis to determine what code may execute while the lock is held.

The refactoring now invokes function `canUseReadLock` (defined in lines 13–19) to determine whether the block b of code protected by the lock is free from non-local side effects, so that a read lock can be introduced. This function, in turn, uses function `nonLocalSideEffects` to determine whether any of the instructions S in b modify non-local state.

Function `nonLocalSideEffects` (lines 20–30) takes as parameters a set S of instructions in some method m and a set P of the relevant parameters of m. As shown in the `nonLocalWrite` function (lines 31–37), a heap write instruction i is deemed non-local iff (1) i writes a static field or (2) i writes an instance field or the array contents of some object o, such that o is reachable (via some sequence of dereferences) from non-local state, defined as a static field or some parameter in P. The reachability check is performed via the `reachableFromNonLocalState` call on line 35 and may be implemented with any conservative may-alias analysis.

The parameter set P is used in `nonLocalSideEffects` to exploit knowledge about purely local objects, which may be safely mutated while a read lock is held. Initially, P contains all formal parameters of the method m containing the protected block b (line 19), as mutations to the state of any of m's parameters may prevent the use of a read lock. When analyzing some method (transitively) called by m, however, we need only consider side effects to any formal parameter f whose corresponding actual parameter a at the caller

```
1: procedure CONVERT TO READ-WRITE LOCK (Field f):
2: assert f has type ReentrantLock
3: change declared type of f to ReentrantReadWriteLock
4: for all assignments to f do
5:    assert f is assigned a new ReentrantLock object
6:    change to assignment of new ReentrantReadWriteLock
7: for all uses u of f do
8:    assert u is call f.lock() or f.unlock()
9:    if u in try-finally construction with body b
        and canUseReadLock(b) then
10:      replace use of f with f.readLock()
11:   else
12:      replace use of f with f.writeLock()

13: function canUseReadLock (Block b):
14: S ← instructions in b
15: if b is in a constructor or method m then
16:    P ← parameters of m, including this if m is not static
17: else
18:    P ← ∅
19: return ¬nonLocalSideEffects(S, P)

20: function nonLocalSideEffects (Set<Instr> S, Set<Param>
    P):
21: for all instructions i ∈ S do
22:    if nonLocalWrite(i,P) then
23:       return true
24:    else if i is a method call then
25:       for all methods m' possibly called by i do
26:          S' ← instructions of m'
27:          P' ← {f : formals of m' | a is actual for f ∧
                    (reachableFromNonLocalState(a, P) ∨
                    canReachNonLocalStateVia(a, P))}
28:          if nonLocalSideEffects(S',P') then
29:             return true
30: return false

31: function nonLocalWrite (Instr i, Set<Param> P):
32: if i writes a static field then
33:    return true
34: else if i writes an instance field or array element then
35:    return reachableFromNonLocalState(basePtr(i),P)
36: else
37:    return false
```

Figure 4: Refactoring CONVERT TO READ-WRITE LOCK

```
public String toString() {
  fLock.lock();
  try {
    StringBuilder result
      = new StringBuilder(f1.toString());
    result.append("\n");
    result.append(f2.toString());
    return result.toString();
  } finally {
    fLock.unlock();
  }
}
```

Figure 5: Example of writes to local objects.

satisfies either: (1) a reachable via dereferences from non-local state or (2) non-local state can be reached via dereferences from a. (We need condition (2) since the callee may read the non-local state from a and then mutate it.) Line 27 in the pseudocode determines the relevant formals using procedures `reachableFromNonLocalState` and `canReachNonLocalStateVia` (both implementable via may-alias analysis). By ignoring writes to local objects via these relevant parameter sets, our algorithm is able to infer more read locks than if it treated all writes as suspect.

Figure 5 gives an example `toString()` method that illustrates the benefits of ignoring writes to local state. The method appends fields `f1` and `f2` to a `StringBuilder` and returns the resulting `String` while holding the `fLock` lock. The calls to `StringBuilder.append()` mutate the state of the `StringBuilder`. If the analysis did not distinguish writes to local state, this mutation would prevent the use of a read lock. However, our analysis is able to show that the `StringBuilder` pointed to by `result` is purely local. Hence, when the append method is analyzed for side effects, the receiver argument is not considered, enabling the analysis to prove that using a read lock is safe. Note that cases like that of Figure 5 arise very frequently in real code, as `String` concatenation in Java is performed via allocating local `StringBuilder` objects and appending to them. Hence, reasoning about local state is essential to our read-write lock refactoring, as failing to infer a read lock in cases like Figure 5 would greatly frustrate users.

For performance, we bound the call depth to which our analysis searches for side-effecting statements (not shown in Figure 4 for clarity). If the call depth exceeds the bound, the analysis conservatively assumes that unanalyzed calls may write to non-local state. Potential targets at virtual calls are computed based solely on the program's class hierarchy; more precise reasoning typically requires computing what code is reachable given certain entrypoints, which is unsuitable for refactoring tools (e.g., when refactoring a library with no client code available).

In our implementation, we use a custom demand-driven may-alias analysis to implement the `reachableFromNonLocalState` and `canReachNonLocalStateVia` procedures. Whole-program pointer analyses often cannot be used in a refactoring tool, due to excessive runtime and their reliance on knowing program entrypoints. Instead, we determine potential aliasing by following interprocedural defs and uses as needed. Again, we set a maximum call depth bound for performance and make pessimistic assumptions about method behavior beyond the bound for soundness.

5. EVALUATION

We have implemented the two refactorings introduced in the previous sections as a plugin for the Eclipse IDE.[5] In this section, we report on an experimental evaluation of these refactorings on real-world benchmarks.

5.1 Implementation Issues

Our implementation of CONVERT TO REENTRANT LOCK closely follows the pseudocode given in Section 3. To determine whether a field f is unshared, it checks the following three conditions:

1. any assignment to f assigns it either `null` or a new object;
2. the value of f is never assigned to a field, passed as an argument to a method or constructor, or returned as a result;
3. no method invoked on f can cause its value to become shared.

[5]The plugin is available for download from `http://progtools.comlab.ox.ac.uk/projects/refactoring/relocker`.

Benchmark	number of mon. actions	FM	CM	TM	involves library	ambiguous mon. action	unmodifiable type	refactorable
`hsqldb`	746	10	22	714	2	25	23	696 (93.3%)
`xalan`	90	7	4	79	4	6	8	72 (80.0%)
`hadoop-core`	412	10	40	362	0	61	22	329 (79.9%)
`jgroups`	440	110	6	324	76	38	59	267 (60.7%)
`cassandra`	62	0	13	49	1	0	0	61 (98.4%)
total	1750	137	85	1528	83	130	112	1425 (81.4%)

Table 2: Evaluation of CONVERT TO REENTRANT LOCK**.**

To check the third condition, we make sure that any method that is invoked on f is *discreet*, meaning it does not synchronize on **`this`**, nor assigns **`this`** to a field, nor passes it as an argument or returns it as a result, and only invokes discreet methods on **`this`**.

Clearly these conditions are sufficient for f to be unshared. While it might be possible to analyze methods for discreetness during the refactoring, it turns out that in real-world code only a few methods, mostly from the collection classes in the standard library, are usually invoked on unshared fields. We hence decided to hardcode in our implementation a list of methods that we manually checked to be discreet, and consider every other method to be indiscreet.

Another pragmatic issue that arises when implementing CONVERT TO REENTRANT LOCK is concerned with the presence of monitor actions in compiled code. For performance reasons, it is infeasible to compute the abstract monitor of every such monitor action (which in particular involves local type inference on bytecode methods). Instead, we decided to only consider the monitor actions arising from compiled synchronized methods, ignoring compiled synchronized blocks and calls to `wait` and `notify`.

In principle, this might lead to unsoundness in situations where a library synchronizes on an object created in application code that is passed to it as an explicit method parameter (i.e., a parameter other than **`this`**). However, such coding practices would give rise to extremely fragile code that could deadlock unexpectedly. We thus felt justified to ignore this potential source of unsoundness, all the more so because a conservative approach based on static analysis would likely be very imprecise and disallow the refactoring in many cases where it is perfectly safe. To the best of our knowledge, our benchmarks do not exhibit this kind of behavior.

Our implementation of CONVERT TO READ-WRITE LOCK closely follows the pseudocode presented earlier (see Figure 4 in Section 4). The side-effect and alias analyses are implemented using WALA [32]. In our experiments, we bound the call depth explored by the main refactoring to 10 and the alias analysis depth for 3; larger bounds yielded little benefit in our experiments.

The implementation relies on specifications of the heap-updating side-effects of certain frequently used methods from the standard Java library, e.g., `equals()` and `hashCode()` from `java.lang.Object`, some methods of class `String` and `StringBuffer`, and several methods from the standard collection classes. These specifications significantly improve the effectiveness of the refactoring, enabling it to skip analysis of common library methods and well-understood methods from the application (e.g., implementations of `equals()`), improving performance and precision. While strictly speaking unsound, pragmatic assumptions like this are common in the literature on static analysis (see, e.g., [24]). We believe that the remote chance of unsoundness is outweighed by the significantly improved effectiveness of the refactoring.

5.2 Evaluation of CONVERT TO REENTRANT LOCK

For the CONVERT TO REENTRANT LOCK refactoring, we measured its applicability on several real-world Java programs by exhaustively applying the refactoring to all built-in monitors, trying to refactor as many of them as possible. Our evaluation aims to answer two basic questions:

- How useful is the proposed classification of abstract monitors, and how many monitors of each kind occur in real-world code?
- How effective is the refactoring, i.e., what percentage of uses of built-in monitors is it able to refactor, and why does it fail in the other cases?

Table 2 shows the results of our evaluation. We ran our refactoring on five benchmarks: two fairly large programs, with the database engine `hsqldb` at about 140 thousand lines of source code (KSLOC[6]) and the XSLT processor `xalan` at 110 KSLOC; and three medium size programs, with the `hadoop-core` component of the Apache Hadoop framework at 74 KSLOC, the `jgroups` toolkit at 62 KSLOC and the distributed database system Apache `cassandra` at 36 KSLOC. Our untuned implementation took no longer than 2 minutes to refactor a particular monitor.

The first data column gives the total number of source-level monitor actions in the program; the next three columns show their classification: "FM" indicates the number of actions whose abstract monitor is an F-monitor, "CM" of those with C-monitors, and "TM" of those with T-monitors.

As it turns out, many monitor actions in real code have nothing to do either with unshared fields or with class objects, the latter two categories often forming a small minority. Nevertheless, it is essential to give unshared fields a special treatment: many unshared fields are of type `Object`; if they were classified as T-monitors, they would effectively prevent us from refactoring any other T-monitor, since our type-based analysis would not be able to exclude the possibility of aliasing.

The next three columns of the table account for those monitor actions that could not be refactored, categorizing them according to the source of failure: the first column gives the number of monitor actions that could not be refactored because this would necessitate refactoring another monitor action which is not from source; the second tallies those monitor actions where another monitor action was encountered that might operate on the same monitor, but does not definitely do so; and the final column shows the number of monitor actions that could not be refactored because the refactoring would have entailed modifying an unmodifiable type declaration.

Monitor actions of the first category are often synchronized methods in user-defined subclasses of the library class `java.lang.Thread`: that class has some synchronized methods which cannot be changed, so neither can the methods in any of its subclasses. For many of the monitor actions in the second category, a more precise analysis would presumably be able to prove that no monitor aliasing is possible and hence allow the refactoring

[6]As determined by David A. Wheeler's 'SLOCCount'.

Benchmark	read-write locks	uses of read lock	uses of write lock	corr. inf. read locks
hsqldb	5	20	51	5 (25.0%)
hadoop-core	1	8	2	8 (100%)
jgroups	1	5	7	5 (100%)
mina	2	5	6	5 (100%)
cassandra	2	13	7	8 (61.5%)
seraph	2	4	5	4 (100%)
total	13	55	78	35 (63.6%)

Table 3: Evaluation of CONVERT TO READ-WRITE LOCK.

to go ahead; nevertheless it is encouraging to see that even a very simple type-based analysis can handle most cases well enough.

The final column gives the number and percentage of monitor actions that could be successfully refactored. Generally, our tool is able to refactor upwards of 80% of all monitor actions, in some cases significantly more. We do not suggest that all these monitor actions *should* be replaced by corresponding uses of reentrant locks: that is not for a refactoring tool to decide. Rather, our tool provides the possibility for the programmer to perform this refactoring successfully in the vast majority of cases at the push of a button.

5.3 Evaluation of CONVERT TO READ-WRITE LOCK

To evaluate the CONVERT TO READ-WRITE LOCK refactoring, we looked at six major applications that already use read-write locks. We then manually refactored them back to reentrant locks, and used our tool to attempt to "re-infer" the original read-write lock usage. Our measure of success for this refactoring is simply how many uses of read locks the refactoring is able to infer correctly. (Inferring write locks is, of course, trivial.)

The results of this experiment are given in Table 3. We use mostly the same benchmarks as above, except for `xalan`, which does not use read-write locks at all; instead, we consider the Apache `mina` network application framework, a medium-sized program of 51 KSLOC, and the J2EE web application security framework Atlassian `seraph`, which consists of only about 5000 lines of source code. Refactoring a reentrant lock to a read-write lock took no more than one minute for any benchmark.

For each benchmark, we give the total number of read-write lock fields in the program. Next, we list the number of uses of the read and write locks, respectively, and finally the number and percentage of correctly inferred read locks. In all cases with write locks, our analysis soundly determined that use of a read lock was unsafe. We were able to infer all read locks for `mina`, `hadoop`, `jgroups` and `seraph` and more than half for `cassandra`, but only 25% for `hsqldb`.

Having inspected the cases where we fail to infer read locks in `hsqldb` and `cassandra` more closely, we believe handling them is beyond any practical analysis for a refactoring. In all of the failing cases, writes to non-local state can actually occur while the read lock is held, assuming the corresponding call targets in our class-hierarchy-based call graph are feasible. Building a more precise call graph via flow analysis is quite difficult since in many cases, client code for library methods using read locks is missing.

`hsqldb` in particular makes fairly sophisticated use of read locks, as fields in certain caches are updated in a racy manner while read locks are held. We consulted the `hsqldb` developers by email about this issue, and they confirmed that such races can in fact occur, but are harmless since the cached data is not mutated at the same time, and hence the result will always be consistent. The kind of global reasoning required to prove the safety of read locks in such cases is clearly beyond the capabilities of our analyses, and it seems likely that any analysis powerful enough to handle this kind of situation would be too heavyweight for use in a refactoring tool.

In contrast to CONVERT TO REENTRANT LOCK, the CONVERT TO READ-WRITE LOCK refactoring hardly ever fails, since it can always just replace uses of the reentrant lock with uses of the write lock. We envision its use as a first step in converting a reentrant lock to a read-write lock: it will consistently update the declaration and all uses, and directly introduce read locks for the simple cases, staying on the safe side and introducing write locks for the trickier ones. It is then up to the programmer to convert those remaining locks to read locks if needed, based on their understanding of the semantics of the program.

5.4 Assumptions and Threats to Validity

Our implementation make two pragmatic, but strictly speaking unsound assumptions. First, we assume that **`synchronized`** blocks and invocations of `wait` and `notify` in compiled methods can be disregarded for analysis purposes.[7] Second, we provide specifications of the non-local side effects of several well-known standard library methods, and assume that all overriding methods conform to these specifications.

These assumptions, motivated and justified in greater detail in Section 5.1, are the only two sources of potential unsoundness. For performance reasons we use imprecise approximations in two other places: we use a built-in table of known discreet methods, and when analyzing code for potential non-local side effects, we bound both the call graph exploration depth and the interprocedural alias analysis depth. In both cases, however, we make conservative assumptions in cases where the approximation fails, which cannot introduce unsoundness.

Perhaps the main threat to the validity of our results is that the benchmark programs may not be representative of all programs; however, we have chosen a set of programs that represents heavily used examples of several major application domains: databases, XML processing, column stores and map-reduce frameworks. Thus, while it may be that other classes of applications could exhibit very different attributes, it seems very likely that, even if that were to be, our results should still apply across a wide range of applications.

6. RELATED WORK

Most previous work on refactoring has concentrated only on sequential programs without considering the implications of concurrency. Early refactoring research studied the mechanics of expressing behavior-preserving transformations for sequential programs using pre- and post-conditions [20, 23] and program dependence graphs [9, 10]. Later work considered various features of sequential programs such as class hierarchies [12, 28, 29], generics [5, 6, 14, 31], design patterns [13], and access modifiers [27]. Other research has focused on refactoring support to adapt programs to evolving libraries [3], combining refactorings with other programs transformations [22], empirical studies [18, 19], and domain-specific languages for specifying refactorings [30].

Schäfer et al. [26] proposed to formulate refactorings in terms of their effect on static semantic dependencies, such as name binding or def-use dependencies. The present authors then showed how this approach can be extended to a concurrent setting by using dependencies to statically capture concurrent behavior [25].

Recently, there has been a lot of interest in refactoring programs to enhance their concurrent behavior. The CONCURRENCER tool of Dig et al. [2] refactors code to make use of `ConcurrentHashMaps` and `AtomicIntegers`, also provided by the `j.u.c` library. Use of such types is another technique for improving scalability by reducing

[7]Recall, however, that compiled synchronized methods are handled soundly.

lock contention. Dig et al.'s RELOOPER tool [4] refactors loops to execute in parallel via the proposed `ParallelArray` class. Recently, the same group have also proposed a technique and a tool for making classes immutable, thereby ensuring their thread safety [15].

Balaban et al. [1] considered the issue of migration from legacy classes (e.g., `Vector`) to functionally equivalent classes that replace them (e.g., `ArrayList`). While this work was primarily concerned with handling incompatibilities in the APIs, some support was provided to ensure that synchronization behavior was preserved, by inserting synchronization wrappers where needed. The REENTRANCER tool transforms sequential programs to be reentrant, enabling safe parallel execution [33]. Finally, several researchers have tackled the problem of implementing refactorings for X10, a Java-based language with sophisticated concurrency support, and have reported promising first results [7, 16].

7. CONCLUSIONS AND FUTURE WORK

The Java class libraries now provide flexible locking constructs that can improve performance by reducing lock contention. However, experimenting with these locks has been difficult as it requires non-trivial code transformations. We have presented algorithms for determining how programs can be refactored to use `ReentrantLocks` and `ReadWriteLocks` instead of built-in monitor locks, and implemented these algorithms in an automated refactoring tool called *Relocker*. In an evaluation on a collection of Java programs, *Relocker* was able to convert over 80% of the monitor locks in these programs into `ReentrantLocks`. Moreover, *Relocker* was able to infer read-locks in most cases where programmers had previously introduced them manually.

Future work includes the design of refactorings for shrinking the regions of code protected by locks, possibly by taking advantage of the ability of `ReentrantLocks` to protect non-block-structured regions. In the same spirit, a future tool could help developers safely downgrade write locks to read locks (a feature supported by `ReadWriteLock`) to further decrease lock contention.

Acknowledgments

We thank the anonymous reviewers and Danny Dig and his colleagues for their helpful comments and suggestions, and Fred Toussi for his explanations of read lock usage in `hsqldb`.

References

[1] I. Balaban, F. Tip, and R. M. Fuhrer. Refactoring Support for Class Library Migration. In *OOPSLA*, 2005.

[2] D. Dig, J. Marrero, and M. D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *ICSE*, 2009.

[3] D. Dig, S. Negara, V. Mohindra, and R. E. Johnson. *ReBA*: *Re*factoring-aware *B*inary *A*daptation of Evolving Libraries. In *ICSE*, 2008.

[4] D. Dig, C. Radoi, M. Tarce, M. Minea, and R. Johnson. RELOOPER: Refactoring for Loop Parallelism. Technical report, UIUC, 2010. http://hdl.handle.net/2142/14536.

[5] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting Java Programs to Use Generic Libraries. In *OOPSLA*, 2004.

[6] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *ECOOP*, 2005.

[7] R. M. Fuhrer and V. Saraswat. Concurrency Refactoring for X10. In *WRT*, 2009.

[8] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[9] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991. Technical Report 91-08-04.

[10] W. G. Griswold and D. Notkin. Automated Assistance for Program Restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3), 1993.

[11] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[12] H. Kegel and F. Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *ICSE*, 2008.

[13] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[14] A. Kieżun, M. Ernst, F. Tip, and R. Fuhrer. Refactoring for Parameterizing Java Classes. In *ICSE*, 2007.

[15] F. B. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Refactoring for Immutability. In *ICSE*, 2011.

[16] S. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards Concurrency Refactoring for X10. In *PPOPP*, 2009.

[17] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*, pages 267–275, 1996.

[18] E. R. Murphy-Hill and A. P. Black. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *ICSE*, 2008.

[19] E. R. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. In *ICSE*, 2009.

[20] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, UIUC, 1992.

[21] Oracle. `java.util.concurrent.locks` API Specification. `http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/package-summary.html`, 2010.

[22] C. Reichenbach, D. Coughlin, and A. Diwan. Program Metamorphosis. In *ECOOP*, 2009.

[23] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, UIUC, 1999.

[24] A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI*, pages 199–215, 2005.

[25] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip. Correct Refactoring of Concurrent Java Code. In *ECOOP*, 2010.

[26] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping Stones over the Refactoring Rubicon. In *ECOOP*, 2009.

[27] F. Steimann and A. Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In *ECOOP*, 2009.

[28] F. Tip. Refactoring Using Type Constraints. In *SAS*, 2007.

[29] F. Tip, A. Kieżun, and D. Bäumer. Refactoring for Generalization using Type Constraints. In *OOPSLA*, 2003.

[30] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A Scripting Language for Refactoring. In *ICSE*, 2006.

[31] D. von Dincklage and A. Diwan. Converting Java Classes to Use Generics. In *OOPSLA*, 2004.

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

[33] J. Wloka, M. Sridharan, and F. Tip. Refactoring for Reentrancy. In *ESEC/FSE*, 2009.