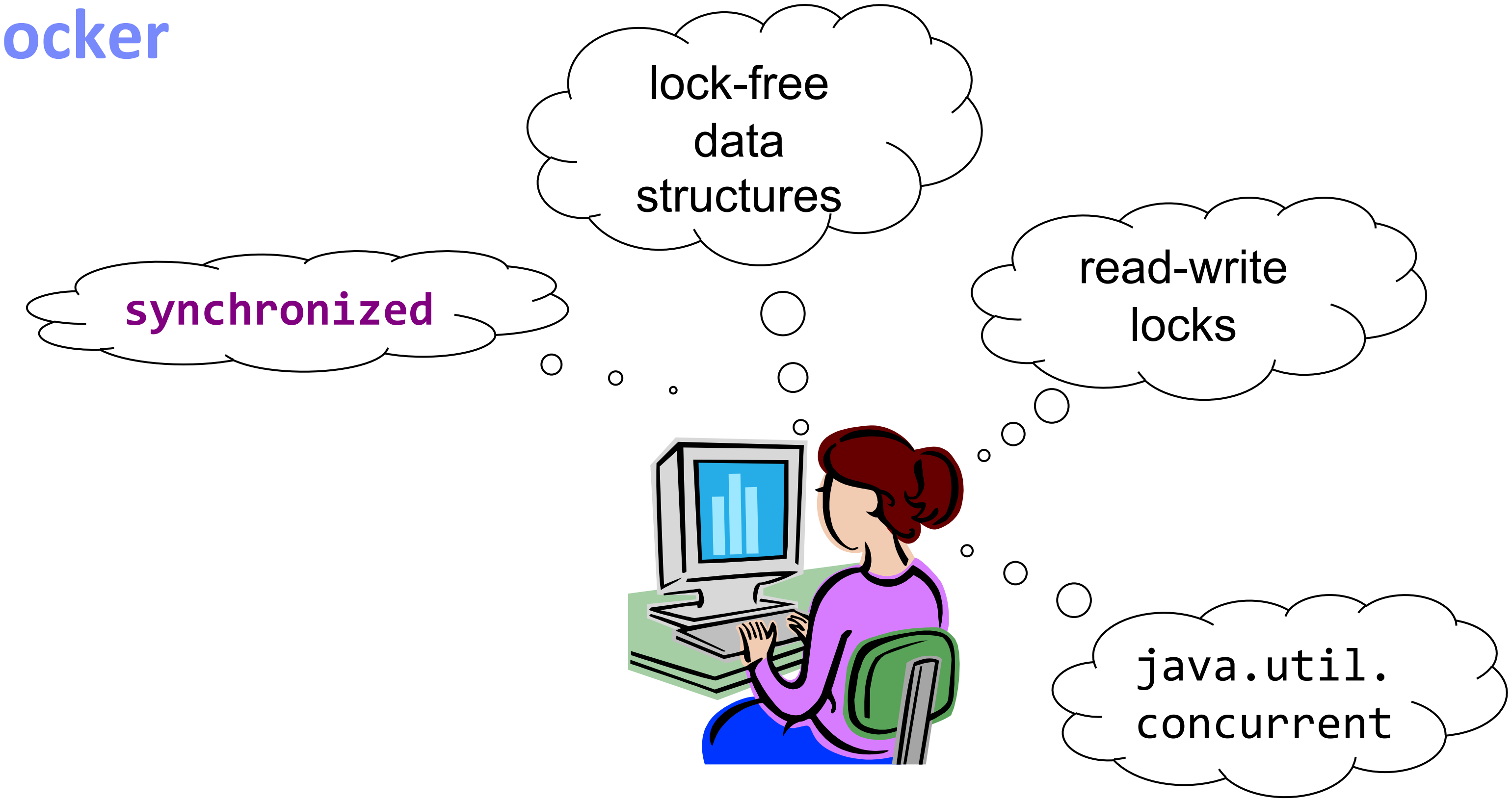
Refactoring Java Programs for Flexible Locking

Max Schäfer[*] Manu Sridharan[†]

Julian Dolby[†] Frank Tip[†]

[*] Oxford University Computing Laboratory
[†] IBM T.J. Watson Research Center

Relocker



tool support for refactoring towards flexible locking constructs

Convert to Reentrant Lock

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

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

Convert to Read-Write Lock

```java
class SyncMap {
  Map map;
  SyncMap(Map map) {
      this.map = map;
  }
  synchronized Object
    put(Object k, Object v){
      return map.put(k, v);
  }
  synchronized Object
    get(Object k){
      return map.get(k);
  }
}
```

```java
class SyncMap {
  ReadWriteLock lock;
  Map map;
  SyncMap(Map map) {
      lock = new ReentrantRWLock();
      this.map = map;
  }
  Object put(Object k, Object v){
      lock.writeLock().lock();
      try { return map.put(k, v);
      } finally {
        lock.writeLock().unlock(); }
  }
  Object get(Object k){
      lock.readLock().lock();
      try { return map.get(k);
      } finally {
        lock.readLock().unlock(); }
  }
}
```

Convert to Reentrant Lock

built-in monitor ⟹ `j.u.c.ReentrantLock`

```
synchronized(e) {
  …
}
```

⟹

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

UNIVERSITY OF OXFORD

IBM

Pseudocode

```
convertToReentrant(synchronized block b):
  create lock field l
  for all synchronized blocks b'
  do
    if(mustAccessSameMonitor(b, b'))
      transform b'          // must be in source code
    else if(mayAccessSameMonitor(b, b'))
      error("ambiguous synchronization")
```

b may access different monitors at different times!

Revised Pseudocode

convertToReentrant(synchronized block b):

  M(b) ⊇ monitors accessed by b ← **abstract monitor**

  create lock field l

  for all synchronized blocks b'

  do

    M(b') ⊇ monitors accessed by b'

    if(**M(b') ⊆ M(b)**)

      transform b'

    else if(**M(b') ∩ M(b) ≠ ∅**)

      error("ambiguous synchronization")

all monitors in M(b) are refactored at once!

Type-based Monitor Abstraction

TM(C): all monitors of objects of class *C* and its subclasses

Transformation:

1. insert (non-static) lock field *f* into *C*
2. rewrite **`synchronized`**`(o)` to `o.f.lock()`

use static fields to handle synchronization on class objects

Not good enough!

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

`Map` is an interface, cannot add field

Not good enough!

```
class C {
  private HashMap m = new HashMap();
  Object get(Object k) {
    synchronized(m) {
      return m.get(k);
    }
  }
}
```

`HashMap` is a library class, cannot modify

Not good enough!

```
class C {
  private HashMap m = new HashMap();
  Object get(Object k) {
    synchronized(m) {
      return m.get(k);
    }
  }
}
```

but field `m` is **unshared**:

The monitor of an object stored in `m` must be accessed through `m` itself.

Field-based Monitor Abstraction

FM(f): all monitors of objects stored in unshared field *f*

Transformation:

1. insert lock field *l* alongside *f*
2. rewrite **`synchronized`**`(e.f)` to `e.l.lock()`

```java
class C {
  Map m = new HashMap();
  Object get(Object k) {
    synchronized(m) {
      return m.get(k);
    }
  }
}
```

```java
class C {
  Map m = new HashMap();
  ReentrantLock l =
    new ReentrantLock();
  Object get(Object k) {
    l.lock();
    try {
      return m.get(k);
    } finally {
      l.unlock();
    }
  }
}
```

Implementation

- unsharedness analysis:
  - field only assigned newly constructed objects
  - local escape analysis
  - handwritten specs for frequently used methods
- monitor enters/exits in bytecode *not* analyzed
  - potential unsoundness
  - but synchronized methods are handled

Evaluation

Benchmark	num of synch	FM	CM	TM	involves library	ambig. lock	unmod. type	success
hsqldb	746	10	22	714	2	25	23	93.3%
xalan	90	7	4	79	4	6	8	80.0%
hadoop-core	412	10	40	362	0	61	22	79.9%
jgroups	440	110	6	324	76	38	59	60.7%
cassandra	62	0	13	49	1	0	0	98.4%
Total	**1750**	**137**	**85**	**1528**	**83**	**130**	**112**	**81.4%**

Convert to Read-Write Lock Pseudocode

convertToReadWrite(ReentrantLock field f):
  make f a ReentrantReadWriteLock
  for all uses u =
    `f.lock(); try { b } finally { f.unlock(); }`
  do
    if (*canUseReadLock*(b, f)) **then**
      replace `f` with `f.readLock()` in u
    else
      replace `f` with `f.writeLock()` in u

Convert to Read-Write Lock Pseudocode

convertToReadWrite(ReentrantLock field f):
  make f a ReentrantReadWriteLock
  for all uses u =
    `f.lock(); try { b } finally { f.unlock(); }`
  do
   if (***canUseReadLock*(b, f)**) **then**
    replace f with `f.readLock()` in u
   else
    replace f with `f.writeLock()` in u

Key to effectiveness

Simplest *canUseReadLock*

canUseReadLock(block b, field f) { **return false** }

Yields sound refactoring, but

no additional concurrency

Ideal *canUseReadLock*

canUseReadLock(block b, field f) {
return true if b ***does not modify state protected by f***, **false** otherwise
}

Maximizes sound read lock usage, but
state protected by locks unknown

Our desired *canUseReadLock*

canUseReadLock(block b, field f) {
  return true if b ***does not modify any thread-shared state***, **false** otherwise
}

(Would ideally limit to state protected by lock, but typically unknown)

Finding shared state mutation is hard

All heap writes mutate shared: ***too imprecise***

```
public int size() {
  int s = map.size();
  System.out.println(“size ” + s);
  return s;
}
```

Heap writes for string concat

Textbook side-effect / escape analysis:

too expensive for refactoring tool

(and may not have the whole program)

Solution: demand-driven analysis

```
foo(x) {
  p = new Object();
  p.f = 3; // local

  y = x.f;
  y.g = 4; // non-local
}
```

Solution: demand-driven analysis

```
foo(x) {
  p = new Object();
  p.f = 3; // local

  y = x.f;
  y.g = 4; // non-local
}
```

Solution: demand-driven analysis

```
foo(x) {
  p = new Object();
  p.f = 3; // local

  y = x.f;
  y.g = 4; // non-local
}
```

Solution: demand-driven analysis

```
foo(x) {
  p = new Object();
  p.f = 3; // local

  y = x.f;
  y.g = 4; // non-local
}
```

- Follows (interprocedural) def-use chains
- Virtual call targets via class hierarchy

Implementation

- **Bound explored call depth for performance**
  - Pessimistic assumptions when depth exceeded
- **Handwritten specs for often-called methods**
  - equals(), hashCode(), toString(), methods on Strings and Collections
  - Significantly improves performance and precision, but possibly unsound (e.g., if equals() mutates protected state)

Results

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%)

- Tried to match manually-added read locks
- **In all failure cases, observed writes to shared state**
  - Races in hsqldb; developers said they were benign
  - Possible call graph imprecision in cassandra; hard to fix

Related Work

- **Refactoring to increase concurrency**
  - **Concurrencer (Dig et al., ICSE09)**: introduce ConcurrentHashMap and AtomicInteger
  - **ReLooper (Dig et al., 2010)**: make array loops parallel
  - **Reentrancer (Wloka et al., FSE09)**: make code reentrant
    - Could be improved with Immutator (previous talk)
- **Safe refactoring of concurrent code**
  - **Balaban et al., OOPSLA05**: add synchronization as needed when updating collection usage
  - **Schaefer et al., ECOOP10**: general techniques for making standard refactorings safe

Conclusions

- Tool support needed for lock construct experimentation
- CONVERT TO REENTRANT LOCK and CONVERT TO READ-WRITE LOCK
  - Can successfully transform real-world code
  - Suitable for interactive refactoring engine
- Possible future work
  - Shrink region for which lock is held, possibly to non-block
  - Help downgrade write lock to read lock

Eclipse plug-in download: http://is.gd/relocker