Views: Specifying Locks by Policy, not Implementation

Patrick Lam
University of Waterloo
http://patricklam.ca

Joint work with Brian Demsky (UC Irvine)
Problem

Current locking mechanisms are hard to use.
Problems

- can’t express why locks exist;
- no way to express fine-grained locking;
- no guarantee that locks are consistently applied.
Raise abstraction level of concurrency control primitives.
Subgoals

Support object-oriented design principles.

Support fine-grained locking at language level.

Support static checking to help catch concurrency bugs.
Array-Based Vector Implementation

```
array: [2, 5, 3, 1, 7, 9]
size: 2
capacity: 5
```
Vector class definition

<table>
<thead>
<tr>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>size : int</td>
</tr>
<tr>
<td>capacity : int</td>
</tr>
<tr>
<td>array : int[]</td>
</tr>
</tbody>
</table>

Rule: must hold owning lock to access “array”.
Basic Approach: Compiler Checks

class Vector {
    public Object get(int i) {
        return array[i]; // must hold appropriate view!
    }
}

// ...

Compiler reports error on attempt to read “array” without appropriate view.
Basic Approach: Compiler Checks

class Vector {
    public Object get(int i) {
        acquire (this@read) {
            return array[i];
        }
    }
}

// ...

Acquiring the “read” view permits access to “array”.
Basic Approach: Code Generation

class Vector {
    public Object get(int i) {
        acquire (this@read) {
            return array[i];
        }
    }
}

class Vector {
    public Object get(int i) {
        synchronized (this$lock0) {
            return array[i];
        }
    }
}

// ...

// ...

Our compiler uses regular locks or read/write locks, as appropriate.
Interface of Vector Example

Our Vector provides the following methods:

- get()
- set()
- capacity(): queries current Vector capacity.
- resize(): changes Vector capacity, re-copying items.
Concrete Example: read view

```java
view read {
  size, capacity, array: readonly;

  incompatible write, resize;

  get(int i) preferred;
  capacity();
}
```

Provides read-only access to fields "size", "capacity" and "array". No thread can hold "read" view on an object while some other thread holds view "write" or "resize" on that object. Methods "get" and "capacity" belong to this view. Calling method "get" will auto-acquire this view.
Concrete Example: read view

```java
view read {
    size, capacity, array: readonly;
    incompatible write, resize;
    get(int i) preferred;
    capacity();
}
```

- Provides read-only access to fields “size”, “capacity” and “array”.
Concrete Example: read view

```java
view read {
    size, capacity, array: readonly;

    incompatible write, resize;

    get(int i) preferred;
    capacity();
}
```

- Provides read-only access to fields “size”, “capacity” and “array”.
- No thread can hold “read” view on an object while some other thread holds view “write” or “resize” on that object.
Concrete Example: read view

```java
view read {
  size, capacity, array: readonly;

  incompatible write, resize;

  get(int i) preferred;
  capacity();
}
```

- Provides read-only access to fields “size”, “capacity” and “array”.
- No thread can hold “read” view on an object while some other thread holds view “write” or “resize” on that object.
- Methods “get” and “capacity” belong to this view.
Concrete Example: read view

```java
1 view read {
2   size, capacity, array: readonly;
3
4   incompatible write, resize;
5
6   get(int i) preferred;
7   capacity();
8 }
```

- Provides read-only access to fields “size”, “capacity” and “array”.
- No thread can hold “read” view on an object while some other thread holds view “write” or “resize” on that object.
- Methods “get” and “capacity” belong to this view.
- Calling method “get” will auto-acquire this view.
Views for Vector Example

- capacity: provides read access to capacity field

All other views provide read access to Vector metadata (size, capacity) and:

- read: read access to Vector contents ("get").
- write: write access to Vector contents ("set").
- xclRead: read access to Vector contents; only one thread can hold xclRead.
- resize: write access to Vector contents and metadata.
Resizing the Vector

```java
public void resize(int newcapacity) {
    Object[] newarray = new Object[newcapacity];
    for(int i=0; i < newcapacity && i < size; i++) {
        newarray[i] = array[i];
    }

    acquire (this@resize) {
        array = newarray; capacity = newcapacity;
        size = (size<newcapacity) ? size : newcapacity;
    }
}
```

1. Copy existing Vector’s contents, holding xclRead view.
2. Switch over the Vector to point to the new copy, holding resize view.
Views Ensure Mutual Exclusion

Calling resize() auto-acquires xclRead view.

- No other thread may write while xclRead view held (xclRead incompatible with write, xclRead, resize).

During the critical switchover phase, no other thread may access the Vector: all views incompatible with resize view.
Ensuring Safe Access to Arrays

We enable developers to encapsulate arrays:

- permits local reasoning about array reads and writes.

In our solution:

- `readonly`, `readwrite` access don’t allow array reference to escape (ensured by a static analysis).
Problem with Unencapsulated Arrays

Unencapsulated arrays have the following (aliasing-related) problem:

```java
Object[] escapeArray() { return array; }
// not protected by any views:
escapeArray()[4] = null;
```

(Assume that `escapeArray()` has full rights to the array.)
Arrays: more details

We provide 5 access descriptions for arrays:

- **fieldreadonly**, **fieldreadwrite**: permit array references to escape;
- **readonly**: permits only reads of the array, e.g. `o.f[3]`, but no copies\(^1\).
- **readwrite**: permits reads and writes, but no copies\(^2\).
- **arraywrite**: enables mutation & unlimited reads of `o.f`.

---

\(^1\) Actually, `Object r = o.f` is allowed, but `r` can’t escape.
\(^2\) Same exception as for read.
Sensible Defaults

Goal: minimize instrumentation overhead.

**Base View**: create a default base view (if not explicitly declared) and populate it with:

1. fields (with readwrite access) that belong to no other view;
2. methods that belong to no other view.

**Constructors**: have full access to the object being constructed.
Views and Inheritance

We permit views in subclasses to be (only) more permissive:

```java
class Parent {
  int f;
  view v { f : readonly; }
}

class Child extends Parent {
  view v { f : readwrite; }
}
```

Developers may extend subclass behaviour without being constrained by superclass implementation.

Note: compiler allocates locks based on actual types, not declared types.
Views: Behind the scenes

- Source code w/views
- View declarations
- Compiler
- Lock allocation
- Java source code
Compiler Checks

We check:

- sanity for view declarations;
- adequacy of view acquisitions.
View Declaration Sanity

Recall: view declarations contain incompatibility declarations. Assume \( v_1, v_2 \) incompatible and \( T_1 \) holds \( v_1 \).

- \( T_2 \) must wait for \( v_1 \) to be released before it may acquire \( v_2 \).

Read-Write Hazard Check: inspect view declarations, e.g.,

```
1 view v1 { f: readwrite; }
2 view v2 { f: readonly; }
```

Since \( v_1, v_2 \) are compatible, \( f \) may be subject to races.

General rule: for \( v_1, v_2 \) compatible, warn about any fields with write access in \( v_1 \) and read or write access in \( v_2 \).

Inheritance Check: views in subclasses may (only) be more permissive.
Compiler Checks: Assignments

We implemented a type system to check the following:

\[ x = y; \quad // \text{assignment to } x \]

- Verify that \( x \) is not a method formal, nor declared with a view type (e.g. `Foo@v1 x`).
- (Can only assign to variables with view types at initial declarations, e.g. `Foo@v1 x = y;`, as long as types match.)
Compiler Checks: Field Accesses

x.f; // read field f

- Verify that all possible views of x allow reads of field f.

x.f = y; // write to field f

- Verify that all possible views of x allow writes to field f.
Compiler Checks: Method Calls

\[ x.m(a_1, \ldots, a_n); \quad // \text{method call} \]
\[ \quad // \text{target: } m(f_1, \ldots, f_n) \]

- Receiver check: Verify that all views of \( x \) contain \( m \), or that \( m \) has a preferred view.
- Argument checks: Verify that each argument \( a_i \) at the call site matches view type of formal \( f_i \) (if applicable).
Must not expose arrays with readonly, readwrite access.

Intraprocedural analysis in two phases: 1) create constraints; 2) verify constraints are respected.

Analysis abstraction contains 3 flags per local variable:

- **NO_ESCAPE**: prohibits e.g. `return x;`
- **NO_STORE**: prohibits e.g. `o.f = x;`
- **NO_WRITE_THRU**: prohibits e.g. `o.f[i] = x;`
// Assume: rw access to f, ro access on g, // no constraints on p
Object[] x = foo(), y = o.g;

p.q = x;

x = o.f;

o.f[4] = y;
if (...)
    return x;

x[2] = new Object();
o.f = x;
Analysis Example

// Assume: rw access to f, ro access on g,
// no constraints on p
Object[] x = foo(), y = o.g;
   // y ∈ NO_ESCAPE, y ∈ NO_STORE, y ∈ NO_WRITE_THRU
p.q = x;
   // y ∈ NO_ESCAPE, y ∈ NO_STORE, y ∈ NO_WRITE_THRU (still)
x = o.f;
   // {x,y} ∈ NO_ESCAPE, {x,y} ∈ NO_STORE, y ∈ NO_WRITE_THRU
o.f[4] = y;
if (....)
   return x;
x[2] = new Object();
o.f = x;
   // {x,y} ∈ NO_ESCAPE, {x,y} ∈ NO_STORE, y ∈ NO_WRITE_THRU
Analysis Example

// Assume: rw access to f, ro access on g,
// no constraints on p
Object[] x = foo(), y = o.g;

    // y ∈ NO_ESCAPE, y ∈ NO_STORE, y ∈ NO_WRITE_THRU
p.q = x; // OK, no constraints on x or p
    // y ∈ NO_ESCAPE, y ∈ NO_STORE, y ∈ NO_WRITE_THRU (still)
x = o.f; // OK, have read access on f
    // {x,y} ∈ NO_ESCAPE, {x,y} ∈ NO_STORE, y ∈ NO_WRITE_THRU
o.f[4] = y; // violates y ∈ NO_WRITE_THRU
if (...) 
    return x; // violates y ∈ NO_ESCAPE
x[2] = new Object(); // OK (x = o.f was rw)
o.f = x; // violates x ∈ NO_STORE
    // {x,y} ∈ NO_ESCAPE, {x,y} ∈ NO_STORE, y ∈ NO_WRITE_THRU
1. Creating constraints

Constraints propagate forward, and arise as follows:

```java
o.f = x; // assume readonly/rw access to f
creates x.NO_ESCAPE, x.NO_STORE.
```

```java
Object[] x = o.f;
creates x.NO_ESCAPE, x.NO_STORE
(for readonly access to f), creates x.NO_WRITE_THRU.
```

```java
return x; // or other statements escaping x
creates x.NO_STORE
```

```java
y = x;
creates x.NO_STORE, y.NO_STORE.
```

We also have rules for `System.arraycopy`. 
2. Checking constraints

```plaintext
o.f = x;
    verify !x.NO_ESCAPE, and if f rw, !x.NO_STORE

Object[] x = o.f;
    verify read-only access on f

return x; // or other statements escaping x
    verify !x.NO_ESCAPE

y = x;
    verify !x.NO.Escape

x[i] = ...;
    verify !x.NO_WRITE_THRU
```
View Inference

Implemented: infer view incompatibility based on hazards.

- If \( \nu_1, \nu_2 \) contain field \( f \), with \( \nu_1 \) granting readwrite
  \( \Rightarrow \) \( \nu_1, \nu_2 \) incompatible.


- Record field reads and writes in \( \text{synchronized()} \) blocks.
- Create view definitions based on accessed fields.
- Convert \( \text{synchronized()} \) blocks into \( \text{acquire()} \) blocks.
Views: Behind the scenes

source code w/views

view declarations

compiler

lock allocation

Java source code
Lock Allocation Example

Compile incompatibility graph (vertices = views, incompatibility = edges, boxes = cliques) into lock allocation.

Use greedy algorithm to approximate clique cover.
Lock Allocation Example

Compile incompatibility graph (vertices = views, incompatibility = edges, boxes = cliques) into lock allocation.

One lock per clique.
No self-compatible views in clique = normal lock.
Lock Allocation Example

Compile incompatibility graph (vertices = views, incompatibility = edges, boxes = cliques) into lock allocation.

Exactly one self-compatible view in clique = read/write lock.
Acquiring a View

Compiler translates view acquisition into lock acquisition.

To acquire a view $v$, acquire the lock corresponding to each clique that $v$ belongs to.

- for a read/write lock, acquire in read mode if the view is self-compatible.
Results

No speedups.

1. Microbenchmarks: views are fast enough.
2. Case studies: views are useful.
Microbenchmarks

Investigate performance of views, compared to naïve locks and hand-coded implementations.

Two benchmarks:
- Red-black tree
- Concurrent hash map

Hardware: dual-processor quad-core Intel Xeon E5520.
Red-black tree performance
Concurrent hash map performance (vs. % writes)

![Graph showing the performance comparison of different hash maps, including java.util.Hashtable (built-in synchronization), ConcurrentHashMap with manual locks, ConcurrentHashMap with views, and java.util.concurrent.ConcurrentHashMap. The graph plots time (ms) against % writes.](image-url)
Concurrent hash map performance (vs. # cores)

java.util.Hashtable (built-in synchronization)
ConcurrentHashMap with manual locks
ConcurrentHashMap with views
java.util.concurrent.ConcurrentHashMap
Case study results

Implemented Views as a Polyglot (extensible compiler framework) extension.


Acquired and ported 4 multi-threaded Java applications:

- Vuze file-sharing client
- Mailpuccino mail client
- TupleSoup database library
- jPhoneLite voIP softphone
Vuze file-sharing client case study

194,000 lines of code total.

We modified the buddy plugin for Vuze, 13,500 lines of code.

Changed two classes:
- BuddyPlugin, 4 views
- BuddyTracker, 9 views
Vuze: BuddyPlugin class

Added 4 views:

- **read_state**: provides read access to 11 fields, incompatible with **write_state**
- **write_state**: provides write access to 11 field, incompatible with **read_state** and **write_state**
- **pd_queue, publish_write_contents**: lock-like, provide read/write access to 1 field, incompatible with self

Note how views protect part of the class’s state and ensure appropriate locks are held.
Vuze: BuddyPluginTracker class

Most interesting locking structure in buddy plugin.

Converted 5 locks into 6 views:

- “this” lock maps to 2 views, “read_internal_state” and “write_internal_state”
- other locks generally map to 1 view

For instance:

- view “online_buddies” provides read/write access to fields “online_buddies”, “online_buddy_ips”.
- view “read_internal_state” provides read access to 12 fields, incompatible with “write_internal_state” and “buddy_peers”.
Mailpuccino graphical mail client

Contains 14,000 lines of code.

Ported the mail folder cache to use views, allowing multiple threads to simultaneously read the message cache.

Created 4 views in all: 2 sets of 2 views each.

1. lookup and modify views: protect the “KeyValues” field (which stores the cache) and its accessor methods.

2. file and index views: protect the cache file and its index. Only one thread may access a file at a time.

Compiler synthesizes 3 locks: 2 normal plus 1 read/write lock.
Mailpuccino graphical mail client: bad code

Benchmark also contains a class MonitoredInputStream.

- Tried to replace two synchronized methods with views.
- Compiler warned that a third unsynchronized method accessed a view protected method.
- Discovery: MonitorInputStream was not thread safe and the synchronized methods were never actually called.
jPhoneLite VoIP softphone

Contains 20,000 lines of code.

Annotated the RTP and RTPChannel classes.

- **readSamples**, **writeSamples**: protect a circular buffer containing incoming data.
  readSamples writes to the buffer metadata.

- **readHostPort**, **writeHostPort**: protect destination information (remoteip, remoteport fields)

Compiler produces 1 read-write lock (host port) and 1 normal lock (samples) for RTP, plus 1 normal lock for RTPChannel.

Also kept the original lock on RTP to protect remaining state.
TupleSoup database library

Contains 6,000 lines of code.
Three index classes: MemoryIndex,PageIndex, FlatIndex.

Original implementation used one lock to protect all indexes.

Created 2 views per index class: access and modifying.
  - multiple threads may hold an access view;
  - only one thread may hold modifying view, and no other thread may access or modify concurrently.

Compiler uses read-write locks to implement these views.

Also converted a cached table (5 locks) to use views.
  - one lock was unnecessary; views made this obvious.
Related Work

- Type systems to ensure absence of data races (Boyapati et al; Abadi et al; Bacon et al)
- Race detection tools (Eraser; Choi et al; Marino et al; RacerX; Warlock; Sema)
- Automatic generation of locking schemes for critical regions (Halpert et al; Emmi et al; Hicks et al; Zhang et al)
Conclusion

Presented the views concurrency control mechanism:

- raise the abstraction level of concurrency control; and,
- enable static checking that can help catch concurrency bugs.

Experience indicates that views are:

- simple to program with;
- support sophisticated fine-grained access control; and
- can detect concurrency bugs.