Views: Object-Inspired Concurrency Control

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

Patrick Lam
University of Waterloo
p.lam@ece.uwaterloo.ca

ABSTRACT

We present views, a new approach to controlling concurrency. Fine-grained locking is often necessary to increase concurrency. Correctly implementing fine-grained locking with today's concurrency primitives can be challenging—race conditions often plague programs with sophisticated locking schemes. Views ease the task of implementing sophisticated locking schemes and provide static checks to automatically detect many data races.

Views consist of view declarations that describe which views of an object may be simultaneously held by different threads, which object fields may be accessed through a given view, and which methods can be called through a given view. A set of view annotations specify which code regions hold a view of an object. Our view compiler performs simple static checks which eliminate many data races.

We have ported three benchmark applications to use views: portions of Vuze, a BitTorrent client; Mailpuccino, a graphical e-mail client; and TupleSoup, a database. Our experience indicates that views are easy to use, make implementing sophisticated locking schemes simple, and can help eliminate concurrency bugs.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Languages, Design, Reliability

Keywords

concurrency, language design, static verification

1. INTRODUCTION

With the wide-scale deployment of multi-core processors, developers must write parallel software to realize the benefits of continued improvements in microprocessors. Using existing concurrency primitives such as locks can be difficult and error-prone, since these primitives force developers to specify implementation, not policy. Currently, developers must manually and painstakingly state how locking is to be implemented in a software system, not why locks exist.

This paper presents a new approach to concurrency control. Instead of providing low-level concurrency primitives, our approach raises the abstraction level of concurrency control to the level of object interfaces. A view specifies both a partial object interface and a list of incompatible object views. A partial object interface lists a subset of the object's methods and fields; a thread must hold the given view to access the part of the object's interface protected by the view. Additionally, the view mechanism provides concurrency control by enforcing incompatibility of views—two views are incompatible if two different threads cannot simultaneously hold the two views on the same object.

Views have two primary benefits. First, views enable developers to easily implement sophisticated concurrency control mechanisms which can maximize an application's concurrency. Views accomplish this goal by providing a simple language mechanism which allows developers to protect subsets of an object's fields and methods. They also provide a uniform mechanism which supports advanced concurrency primitives such as read-write locks. Second, views can statically detect many data races. Our view compiler can analyze view specifications and objects' uses of views to detect and warn about possible race conditions and unprotected field and method accesses.

1.1 Contributions

This paper makes the following contributions:

- **View Concept:** It presents a new concurrency primitive that expresses concurrency control as acquisition of partial object interfaces, or views. Views provide a simple, natural abstraction that support a wide range of advanced locking approaches. Views make the connection between the concurrency primitive and the data it protects explicit.
- **Automatic Lock Synthesis:** It presents an algorithm that uses standard locks to implement views. Our compiler uses a greedy algorithm to synthesize an optimized implementation of views using locks.
- **Static Checking:** It presents several static checks that automatically detect possible concurrency bugs. These checks can detect view specifications that allow race conditions on data. These checks can also detect field and method accesses in which the developer neglected to acquire the proper object view.

ICSE '10, May 2-8 2010, Cape Town, South Africa

- **Experience with Views:** It presents our experience porting three significant benchmarks to use views. This experience indicates that it is relatively simple to use views, that views make supporting advanced locking straightforward, and that views can help to statically detect potential concurrency bugs.

The remainder of the paper is structured as follows. Section 2 presents an example to illustrate our approach. Section 3 presents the view extensions to Java. Section 4 describes how we compile views. Section 5 presents our experience using views with three existing applications. Section 6 discusses related work. Finally, Section 7 concludes.

2. EXAMPLE

We present an example that illustrates the use of views. Figure 1 presents a single-threaded implementation of the `Vector` class. This `Vector` class contains a `set()` method to set elements of the vector, a `get()` method which returns the current value of an element, and a `resize()` method which resizes the `Vector`. We omit `remove()` for space reasons; its implementation would mirror that of `resize()`.

Views consist of two parts: view declarations, which identify the members of each view, and view annotations to Java source code, which acquire views as needed throughout the implementation. Figure 2 presents modifications to lines 28 through 31 of the existing `Vector` code to support views. Figure 3 presents view declarations for `Vector`.

2.1 View Annotations

Our system allows threads to acquire views in two ways: 1) a thread may explicitly acquire a view using the `acquire` statement, and 2) a thread may implicitly acquire a view by calling a preferred method.

The statement `acquire(this@resize)` in Figure 2 causes the thread to acquire the `resize` view of the object referenced by `this` before executing lines 29–30 and then to release this view in line 31. Note how `acquire` generalizes Java's `synchronized` construct. The relevant view declaration (see below) explains what the view protects.

When a thread makes a call to a preferred method, such as `get()` for the `read` view, without already holding a view that provides access to that method, the thread will automatically acquire the appropriate view and then execute the method. A non-preferred method is only callable by threads that already hold a view that contains the method.

2.2 View Declarations

Figure 3 declares five views: `read`, `write`, `xclRead`, `resize` and `capacity`. Each view, except `resize` and `xclRead`, corresponds to a method of `Vector`, and states the fields and methods required to execute that method. The views `xclRead` and `resize` support the `resize()` operation's two phases—an exclusive-read phase, in which `resize()` copies the `Vector`'s contents, followed by the `resize` phase, which atomically writes to the `Vector`.

View declarations include a view's name and its body. Figure 3 begins with the `read` view. A view body first lists views that are incompatible with the current view. For example, line 2 declares that the `read` view is incompatible with the `write` and `resize` views: no thread may acquire an object's `read` view while any other thread holds the `write` or `resize` views of that object.

```
1 public class Vector {
2  int size;
3  int capacity;
4  Object[] array;
5
6  public Vector() {
7   size = 0; capacity = 10;
8   array = new Object[capacity];
9  }
10
11 public Object get(int i) {
12  if (i < size) return array[i];
13  else return null;
14 }
15
16 public void set(int i, Object o) {
17  if (i < capacity()) {
18   array[i] = o;
19   size = ((i+1)>size) ? (i+1) : size;
20  }
21 }
22
23 public void resize(int newcapacity) {
24  Object[] newarray = new Object[newcapacity];
25  for(int i=0; i < newcapacity && i < size; i++) {
26   newarray[i] = array[i];
27  }
28
29  array = newarray; capacity = newcapacity;
30  size = (size<newcapacity) ? size : newcapacity;
31
32 }
33
34 public int capacity() {
35  return capacity;
36 }
37 }
```

Figure 1: Sequential Vector Example.

```
28 acquire (this@resize) {
29   array = newarray; capacity = newcapacity;
30   size = (size<newcapacity) ? size : newcapacity;
31 }
```

Figure 2: Changes to Vector to support views.

The view's body also contains the view's field and method declarations. A field declaration begins with a comma-separated list of fields followed by an access description. Field access descriptions are one of `none`, `readonly`, or `readwrite`. Line 3 declares that threads holding the `read` view of a `Vector` object may read its `size`, `capacity`, and `array` fields. Note that the `readonly` declaration ensures read-only access to the field in the same sense as the Java `final` modifier: it does not prevent line 18 of `Vector` from writing to the `Vector`'s underlying `array` object, but only prevents writes of the `array` field itself. A method declaration gives the method's name and the types of its parameters, optionally followed by the keyword `preferred`. Line 4 declares that the `read` view contains the `get()` method with an integer parameter as a preferred member.

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

view write {
  incompatible read, write, resize, xclRead;
  size: readwrite;
  capacity, array: readonly;
  set(int i, Object o) preferred;
  capacity();
}

view xclRead {
  incompatible write, resize, xclRead;
  size, capacity, array: readonly;
  capacity();
  resize(int i) preferred;
}

view resize {
  incompatible read,write,resize,capacity,xclRead;
  size, capacity, array: readwrite;
}

view capacity {
  incompatible resize;
  capacity: readonly;
  capacity() preferred;
}
```

Figure 3: View Declarations for Vector Example.

All classes contain a `base` view, which is usually implicit. The `base` view contains methods and fields which may be accessed without acquiring any view. An implicit `base` view contains all methods and fields not declared in other views. However, developers may also explicitly declare a `base` view, in which case the compiler includes only fields and methods declared to belong to the `base` view. The `base` view is important for supporting object inheritance (see Section 4.2).

2.3 Checking Views

We have implemented an extension to the Polyglot extensible compiler framework [14] to support view annotations, prevent incorrect accesses to view-protected object interfaces, and generate executable code from the view-annotated sources. The compilation process proceeds in three steps. First, the compiler verifies that a program properly uses view declarations, as described below. Next, it uses the view declarations to synthesize a lock allocation: the acquisition of each view corresponds to the acquisition of a set of locks. Finally, it uses the lock allocation to generate code.

We next describe how our view compiler works on our `Vector` example on a method-by-method basis. The compiler grants constructors full access to objects. We expect developers to follow the standard practice of not exposing the object being constructed in the constructor.

The compiler next verifies that the `get()` and `set()` methods respect the view declaration. The compiler observes that the `get()` method accesses the `size` and `array` fields of the `this` object. Both of the fields have readonly access in the `read` view. Because the `get()` method belongs only to the `read` view, `this` must have the `read` view inside `get()`, so the compiler accepts these reads of `size` and `array`. The fact that `get()` is a preferred method is irrelevant to checking the implementation of `get()`—it only affects callers to `get()`, which will automatically acquire the `read` view if they do not already possess it. The verification of `set()` proceeds similarly. However, the compiler also checks that the `write` view possesses write permissions for the `size` field. (Recall that `set()` does not require write permissions to `array` because it is not assigning to the `array` field itself, only to the array object.) Additionally, because `set()` calls the `capacity()` method, the compiler checks that the `write` view contains the `capacity()` method. All checks succeed in our example.

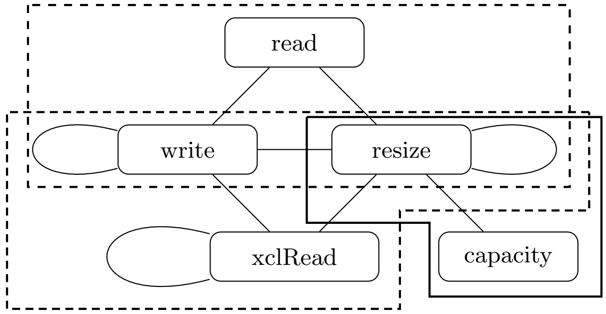

Figure 4: Incompatibility Graph G for `Vector`.

We finally discuss how the compiler verifies the `resize()` method. Note that we chose not to add the `resize` method to the `resize` view. Because `resize()` belongs to the `xclRead` view, the compiler permits the read of field `array` on line 26[1]. The method then explicitly acquires the `resize` view on line 28 of the modified version of `Vector`, granting it permission to write to the `array`, `capacity`, and `size` fields. No other thread may execute any method of `Vector` in parallel with the `resize` view—a thread attempting to access the `Vector` must wait until the `resize` completes.

2.4 Code Generation

To generate code, the compiler must be able to reason about relationships between views, since these relationships determine the set of locks that it must create. It therefore starts by generating a view incompatibility graph. Figure 4 presents the incompatibility graph G for our running example. Graph vertices represent views, while edges between two views indicate that they are incompatible. Dotted lines represent cliques. The edge in G between the `read` vertex and the `write` vertex implies incompatibility of the `read` and `write` views.

Given an incompatibility graph, the lock synthesis algorithm allocates locks by finding a clique covering of the graph: we will associate a lock with each clique. To acquire a view, a thread must acquire locks for all cliques that the view belongs to. The compiler uses read-write locks[2]

---

[1] The compiler correctly displays an error message if `resize()` does not belong to any view granting access to `array`, as in an earlier version of this paper.

[2] A read-write lock [12] can be held by any number of threads in read mode but by only one thread in write mode.

when a clique has exactly one view v which is compatible with itself. Such a situation indicates that v allows concurrent access to the resource being protected (corresponding to the read mode of the read-write lock), while any views v' in the same clique require exclusive access to the resource (write mode). If no views in a clique are compatible with themselves, the compiler uses an ordinary (exclusive) lock.

In our example, the three cliques $C_1 = \{\texttt{read}, \texttt{write}, \texttt{resize}\}$, $C_2 = \{\texttt{write}, \texttt{resize}, \texttt{xclRead}\}$, and $C_3 = \{\texttt{capacity}, \texttt{resize}\}$ cover the graph G. The compiler therefore generates three locks, ℓ_1, ℓ_2, and ℓ_3, one per clique. Cliques C_1 and C_3 contain exactly one view which is compatible with itself, so the compiler uses read-write locks for them. A thread may acquire the `capacity` view by acquiring ℓ_3 in read mode, since `capacity` is compatible with itself; similarly, it may acquire `read` by acquiring ℓ_1 in read mode. A thread may acquire `write` by acquiring ℓ_1 in write mode (since `write` is incompatible with itself) as well as the ordinary lock ℓ_2. To acquire the `resize` view, a thread must acquire write locks on both ℓ_1 and ℓ_3, plus ℓ_2.

The compiler generates code by applying the lock allocation to the view acquisition statements. Intuitively, the compiler will translate a statement like `acquire(this@resize)` into a virtual call to a method on `this` which acquires the `resize` view by requesting the proper locks as per the lock allocation; the virtual call ensures that the thread gets the appropriate locks for the runtime type of `this`, in the presence of inheritance.

To handle preferred methods, the compiler generates a wrapper for the method which requests the view and delegates to the original implementation. In our example, the compiler renames the preferred method `get()` to `get$view()` and generates a new wrapper `get()`, which will hold the `read` view for the duration of the call to `get$view()`. Should a caller to `get()` already hold the `read` view, the compiler simply generates a call to the original method `get$view()` instead of calling the wrapper.

3. VIEW LANGUAGE EXTENSIONS

Figure 5 presents the grammar for view declarations, while Figure 6 presents the syntax extensions to Java for view annotations. As seen in Section 2, view declarations contain a list of incompatible views followed by a list of view members, which may be fields or methods. Field members have associated access descriptions (`none`, `readonly` or `readwrite`). Developers must unambiguously identify methods which belong to a view, and may optionally specify that a method is preferred for a view. We support two kinds of view annotations in Java code: 1) types may be decorated with views (i.e. `Vector@get`); and 2) our new `acquire` statement generalizes Java's `synchronized` statement.

4. COMPILING VIEWS

We next describe in detail how we type check views, check consistent use of views, and automatically generate a locking strategy that enforces view incompatibility constraints.

viewDecl := `view` *name* { *incompDecl*
fieldMethodDecls }
incompDecl := `incompatible` *fieldList*;
fieldList := *fieldList*, *field* | *field*
fieldMethodDecls := *fieldMethodDecls*, *fieldMethodDecl* |
fieldMethodDecl
fieldMethodDecl := *fieldDecl* | *methodDecl*
fieldDecl := *fieldList* : *accessdesc*;
accessdesc := `none` | `readonly` | `readwrite`
methodDecl := *name*(*formallist*) *optpreferred*;
optpreferred := `preferred` | ε

Figure 5: View Declaration.

viewtype := *typename*@*viewname*
formal := … | *viewtype varname*
varDecl := … | *viewtype varname*
statement := … | `acquire`(*varname*@*viewname*) *block*

Figure 6: View Annotations.

4.1 View Types

We have extended the Java type system to support view types for method parameters and local variable declarations. A view type consists of a pair of a reference type and view. For example, the view type `Vector@write` indicates a reference to a `Vector` object for which the executing thread holds the `write` view. The type checker does not allow a local variable or a method parameter with a non-base view type to be re-assigned to reference a different object.

The view type of the `this` variable of a virtual method m is equal to the set of views that contain the method m. The type checker must ensure that fields and methods accessed through the `this` variable are permitted by all views that declare the method.

Both the left and right hand sides of assignments to local variables or method formal parameters with view types must have the exact same view type. New views of an object can only be acquired through an explicit `acquire` or through an implicit acquisition via a call to a preferred method of a view. A method may not have a view type as its return type, nor may fields or arrays have view types. Collectively, these constraints ensure that threads cannot hold a view reference to an object after the release of a view acquired through an acquire statement or a preferred method.

4.2 Static Checks

Compilation begins with several static checks on the view specifications, field accesses, and method calls. Our compiler performs the following checks on the view declarations:

- **Read/Write Hazards on Fields:** For each pair of compatible views (v_1, v_2) and each field f, the compiler flags the field f if v_1 has write access to the field f and v_2 has read or write access to f. If a view is compatible with itself, this check flags all fields that are declared readwrite. Uncontrolled access to flagged fields may lead to race conditions. However, we anticipate that developers may choose to use external locks or other mechanisms to protect such fields. The compiler therefore only produces warning messages for the read/write hazards that it detects.

- **Field Read Checks:** For each field read `x.f`, the compiler checks that all possible views of the receiver expression `x` allow reads of field `f`.

- **Field Write Checks:** For each field write `x.f = y`, the compiler checks that all possible views of the receiver expression `x` allow writes of field `f`.

- **Method Call Checks:** For each method call site `x.m(a`$_1$`, ..., a`$_N$`)` to method `m(f`$_1$`, ..., f`$_N$`)`, the compiler checks that each argument `a`$_i$ at the call site matches the view type of the corresponding method formal parameter `f`$_i$, if `f`$_i$ has a view type. The compiler also checks that the view of the reference to the receiver object `x` contains `m` or that `m` has a preferred view.

- **Assignments:** The compiler checks that the program does not make assignments to local variables or method formal parameters with view types other than at their initial declarations.

- **Field Inheritance Check:** The compiler ensures that an object's fields cannot be accessed through upcasts, in violation of view constraints. To ensure field access safety, the compiler checks that if a field f is declared in a super class of C and is a member of a view v in the super class, then field f must be a member of view v in class C, with at least as permissive access.

- **Method Inheritance Check:** The compiler must ensure that methods cannot be accessed through upcasts in violation of view constraints. To ensure method invocation safety, the compiler checks that if a method m is declared in a super class of C and is a member of a view v in the super class, then method m must also be a member of view v in class C, with at least as permissive access. We make an exception to this check when v is the base view, if m has a preferred view in class C. Note that if a method m is declared in an interface that class C implements, the method m must either be in the base view of class C or have a preferred view in class C.

  It is possible that a call to method $o.m()$ may occur such that the declared type of o would require acquiring a preferred view to call $m()$, but the run-time type of o indicates that the executing thread must already hold the appropriate view. In this case, a dynamic check would avoid needlessly acquiring the preferred view.

4.3 Lock Synthesis

We next describe how we synthesize a locking strategy that enforces the view incompatibility specification. For each class, the lock synthesis algorithm begins by constructing an undirected view incompatibility graph G. The graph G contains a vertex v for each view in c. For each pair of views v_i and v_j, if v_i lists v_j as incompatible, or v_i lists v_j as incompatible, G contains an edge between v_i and v_j.

Consider a subgraph G_C of G that is a clique—that is, G_C contains edges between every pair of vertices in G_C. One lock can enforce all of the view incompatibility constraints between views in G_C. Because views can be incompatible with themselves, self-edges may occur in the view incompatibility graph. We handle self-edges by using different kinds of locks. If all views but one in the clique have self-edges, we use an implementation of a reentrant read-write lock for the clique; we identify the read mode of the read-write lock with the view with no self-edges, and the write mode with all other views in the clique. This corresponds to the situation where any number of threads may hold view v with no self-edges, but only one thread may hold a view v' with self-edges or any of the views v'' that are incompatible with v'. If all views in the clique contain self-edges, then we use the normal reentrant lock class from `java.util.concurrent.locks`. If more than one view in the same clique lacks a self-edge (which we expect to be rare in practice—views without self-edges typically only read data, so two views without self-edges should typically not conflict with each other), we would use a generalized implementation of a read-write lock which would permit multiple mutually-incompatible read locks and a single write lock.

The lock synthesis algorithm computes a clique cover of G. Minimizing the number of cliques in the cover minimizes the number of locks we must generate and the number of locks that must be acquired in a view. However, finding a minimum clique covering for a graph is an NP-complete problem [10]. We therefore use a greedy algorithm to compute a non-minimal clique covering in polynomial time. Our greedy algorithm selects an uncovered edge to cover to start the clique and adds vertices that will cover other uncovered edges. We expect that, in practice, many view incompatibility specifications will be simple enough that our greedy algorithm will generate a minimal covering.

4.4 Acquiring Views

We next describe how the compiled application acquires and releases views at runtime. For each view, the compiler generates three view acquisition methods: the `tryacquireView` method tries to acquire the view, the `acquireView` method acquires the view, and the `releaseView` method releases the view.

To acquire view v, a thread must acquire all of the locks for v. If v has a self edge in the incompatibility graph, the thread must acquire all readwrite locks in write mode and lock the normal reentrant locks. If v does not have a self edge, the thread must acquire all locks, which will be read-write locks, in read mode. The `tryacquireView` method tries to acquire each lock. If it successfully acquires all locks, it returns `true`. If it fails to acquire any of the locks, it releases the locks it has already acquired, and returns `false`.

The `acquireView` method must block until it can ac-

```
public void acquireView() {
  int startindex = 0;
  while (true) {
    // Block on the first lock
    switch (startindex) {
    case 0:
      lock0.lock();
      break;
     ...
    case n-1:
      lock(n-1).lock();
      break;
    }
    // Try to acquire the rest of the locks
    int i;
    loop:
    for (i=1; i<n; i++) {
      if ((++startindex) == n)
        startindex = 0;
      switch (startindex) {
      case 0:
        if (!lock0.trylock())
          break loop;
        break;
        ...
      case n-1:
        if (!lock(n-1).trylock())
          break loop;
        break;
      }
    }
    // Return if we hold all locks
    if (i == n)
      return;
    // Release locks if we failed to get one
    int unlockindex = startindex;
    for(; i>0; i--) {
      if ((--unlockindex) < 0)
        unlockindex = n-1;
      switch (unlockindex) {
      case 0:
        lock0.unlock();
        break;
      ...
      case n-1:
        lock(n-1).unlock();
        break;
      }
    }
    // Repeat, trying to first blocking-acquire
    // the lock that we failed to get this time.
  }
}
```

Figure 7: Locking Code to Acquire A View.

quire a view. To avoid the potential for internal deadlocks, the thread cannot hold any of the component locks while blocking. Figure 7 presents an example of an acquire method that our compiler generates for `n` component locks. Conceptually, the `acquireView` method arranges the locks in a circular list. It locks the first component lock in the list, waiting until this lock becomes available. It then tries to lock the remaining component locks without blocking. If it fails to acquire any of these locks, it releases all of the locks and then repeats the process starting with the lock it failed on. Once it acquires all of the locks, it has acquired the view and returns to the caller. Of course, our compiler generates optimized methods for the single-lock case.

Releasing views is straightforward: the `releaseView` methods simply releases all locks corresponding to a view.

4.5 Simultaneously Acquiring Multiple Views

Our language supports simultaneously acquiring multiple views. We expect that developers will find this mechanism useful for locking multiple shared data structures while avoiding the possibility of deadlock. The generated code for acquiring multiple views would use the same basic strategy as the code in Figure 7 does on component locks, but instead uses this strategy on views.

4.6 Defaults

We have carefully designed the defaults for views to minimize instrumentation overhead. Our compiler automatically generates the base view if the developer does not explicitly declare a base view, according to the following rules:

1. A field is present in the base view with `readwrite` access if no other view declares that field.
2. A method is present in the base view if no other view declares that method.

Object constructors often write to many object fields that would be protected by views and call methods that require access to views. If treated like other methods, object constructors would have to acquire a number of views to access these fields. However, it is relatively rare for object constructors to make the object being constructed accessible to other threads before the constructor exits. Our implementation therefore allows the constructor to access fields and methods of the object being constructed without holding the necessary views. We believe that this is a reasonable tradeoff between usability and detecting possible races.

5. EXPERIENCE

We next discuss our experience adding views to several applications: Vuze, a file-sharing (BitTorrent) client; Mailpuccino, a graphical e-mail client; and TupleSoup, a database.

5.1 Methodology

We have developed a prototype implementation of views as an extension to the Polyglot extensible compiler infrastructure [14]. The source code for our extension is available at `http://demsky.eecs.uci.edu/views/`.

5.2 Vuze Buddy Plugin

Our first benchmark is a subsystem of the open-source Vuze file-sharing client. The source distribution of Vuze is available at `http://azureus.sourceforge.net`. While Vuze contains 194,000 lines of code in all, we chose to concentrate on the buddy plugin of Vuze, which consists of 13,500 lines of code. This plugin is implemented in the `com.aelitis.azureus.plugins.net.buddy` package.

Parts of the buddy plugin contain a rich locking structure. After inspecting the code, we chose to annotate the `BuddyPluginTracker` and `BuddyPlugin` classes. The

```
view read_state {
  incompatible write_state;
  current_publish, latest_publish, buddies,
    buddies_map, config_dirty,
    republish_delay_event, last_publish_start,
    unauth_bloom, ygm_unauth_bloom,
    bogus_ygm_written, write_bogus_ygm: readonly;
}

view write_state {
  incompatible read_state, write_state;
  current_publish, latest_publish, buddies,
    buddies_map, republish_delay_event,
    last_publish_start, unauth_bloom,
    ygm_unauth_bloom, config_dirty,
    bogus_ygm_written, write_bogus_ygm: readwrite;
}

view pd_queue {
  incompatible pd_queue;
  pd_queue: readwrite;
}

view publish_write_contacts {
  incompatible publish_write_contacts;
  publish_write_contacts: readwrite;
}
```

Figure 8: Views for `BuddyPlugin` class.

other classes in the plugin use locking solely to protect data structure accesses: before an access to a non-thread-safe data structure (typically a `Map` or `List`), Vuze acquires the lock on that data structure. Views interoperate smoothly with ordinary Java `synchronized` statements implementing such simple locking strategies.

BuddyPlugin annotations.

We added 4 views to `BuddyPlugin`: general read and write views `read_state` and `write_state`, for mutable fields previously protected by the lock on the `BuddyPlugin` object itself (i.e. `synchronized(this)`), as well as views to protect the `pd_queue` and `publish_write_contacts` data structures. Our compiler found a few field reads that were inconsistently unprotected in the original code.

Our change preserves the existing lock structure and also provides static guarantees that the program doesn't attempt to access protected state without the protecting lock.

BuddyPluginTracker annotations.

We found that the `tracker.BuddyPluginTracker` class contained the most interesting locking structure in the buddy plugin. This class contains 5 different locks: `online_buddies`, `actively_tracking`, `tracked_downloads`, `buddy_peers`, and on the `this` object. We carefully studied the fields that the class accessed under each lock and encoded this information in our view declarations.

Figure 9 presents the view declarations for the `tracker.BuddyPluginTracker` class. We converted the 5 locks into 6 views, splitting accesses to `this` into read-only and read-write views `read_internal_state` and `write_internal_state`, respectively, and changing the other locks into views.

The `actively_tracking` view protects accesses to the `actively_tracking Set`. Its access pattern is similar to that of the other data structures in the buddy plugin.

The `online_buddies` view protects two correlated data structures: the `online_buddies Set` and the `online_buddy_ips Map`. Our view annotations therefore express the formerly-implicit connection between the `online_buddies` lock and the `online_buddy_ips` data structure and statically ensure that the program always follows the proper locking discipline.

The `tracked_downloads` field protects six related fields, including two sets and two maps. In the original version of the `BuddyPluginTracker`, the application always acquired the `tracked_downloads` lock before accessing any of these fields.

Finally, the three views `read_internal_state`, `write_internal_state` and `buddy_peers` all protect miscellaneous internal state of the `BuddyPluginTracker`. Both the `write_internal_state` and `buddy_peers` views provide write access to different parts of the tracker. The `read_internal_state` view is not incompatible with itself, so multiple threads may simultaneously read internal state. Each of the write views is incompatible with itself and with the `read_internal_state` view.

We found that views enable developers to confidently use fine-grained concurrency patterns. Using the view declarations, our compiler statically verifies that the code always acquires the appropriate locks.

5.3 Mailpuccino

Mailpuccino is an open-source graphical mail client written in Java that supports the POP3 and IMAP protocols. Mailpuccino is available at `http://www.kingkongs.org/mailpuccino/`. It contains over 14,000 lines of code.

Mailpuccino maintains separate cache data structures for the message headers, message flags, message parts, and the message structure. The locking for the original cache objects used synchronized methods. The original coarse-grained locking structure only allowed one thread to read from the message cache at a time.

Figure 10 presents the views that we wrote for Mailpuccino's `Cache` object. We created four views in all, belonging to two sets of two views each.

The first set of views includes the `lookup` view and `modify` view for the Mailpuccino cache. The `lookup` view provides read-only access, enabling methods to safely read the cache, while the `modify` view provides read-write access, allowing methods to safely modify the cache. Multiple threads may simultaneously read from `Cache` objects, so the `lookup` view is compatible with itself. However, while any thread is modifying the `Cache` object, no other threads can safely access that `Cache` object at the same time. Therefore, the `modify` view is incompatible with both itself and the `lookup` view. Note that our use of views enables the `Cache` object to potentially support multiple simultaneous `lookup` operations.

The second set of views includes the `file` and `indexfile` views. Each cache is backed by two files: the `DataFile` file and its index, `IndexFile`. Cache misses are served from these files. While the `lookup` view conceptually protects these accesses and prevents simultaneous writes, Java's `RandomAccessFile` object does not support atomic reads from a specific file offset, so Mailpuccino performs a seek followed by a read. We must therefore ensure

```
view actively_tracking {
  incompatible actively_tracking;
  actively_tracking: readwrite;
}

view online_buddies {
  incompatible online_buddies;
  online_buddies, online_buddy_ips:
    readwrite;
}

view tracked_downloads {
  incompatible tracked_downloads;
  tracked_downloads, last_processed_download_set_id,
    last_processed_download_set, download_set_id,
    full_id_map, short_id_map: readwrite;
}

view read_internal_state {
  incompatible write_internal_state, buddy_peers;
  online_enabled, old_plugin_enabled,
    plugin_enabled, old_tracker_enabled,
    tracker_enabled, old_seeding_only, seeding_only,
    consecutive_fails, last_fail, network_status,
    buddy_send_speed, buddy_receive_speed: readonly;
}

view write_internal_state {
  incompatible read_internal_state,
    write_internal_state, buddy_peers;
  online_enabled, old_plugin_enabled,
    plugin_enabled, old_tracker_enabled,
    tracker_enabled, old_seeding_only, seeding_only,
    consecutive_fails, last_fail: readwrite;
}

view buddy_peers {
  incompatible read_internal_state, buddy_peers,
    write_internal_state;
  seeding_only: readonly;
  buddy_peers, buddy_stats_timer, network_status,
    buddy_send_speed, buddy_receive_speed:
    readwrite;
}
```

Figure 9: Views for `BuddyPluginTracker` class.

that no other thread accesses the file object between the seek and the read operations. To do so, we created two more views to protect the file objects. Only threads which have acquired these self-incompatible views may access the fields that reference the corresponding files. This ensures that only one thread may seek and read from a file at a time. While we have described our changes to `Cache`, we also modified the `MsgPartsCache` class in a similar fashion.

We next modified the synchronized methods in the `MonitoredInputStream` class to use views. This class contained two synchronized methods: the `mark` method and the `reset` method. The “synchronized” annotations led us to believe, at first, that the class was designed to be safely shared between threads. The `mark` and `reset` methods access only two fields: `MarkedBytesRead` and `BytesRead`. We wrote a view that allowed access to these fields and added the `mark` and `reset` methods to the view.

At this point, we believed that we had distilled `MonitorInputStream`’s old synchronization pattern into views. We therefore attempted to compile the modified class. Surprisingly, the compiler threw error messages warning that `MonitorInputStream`’s `read` method accesses the `ByteRead` field without holding an appropriate view. However, the `read` method contained no synchronization!

```
view lookup {
  incompatible modify;
  KeyValues: readonly;
  getAsByteArray(Object Key) preferred;
  get(Object key) preferred;
  flush() preferred;
  getKeys() preferred;
  close() preferred;
}

view modify {
  incompatible modify, lookup;
  KeyValues: readwrite;
  put(Object Key, Object Value) preferred;
  remove(Object Key) preferred;
  keepOnlyThese(Vector Keys) preferred;
  compact() preferred;
  getAsByteArray(Object Key);
}

view file {
  incompatible file;
  Data: readwrite;
  DataFile: readonly;
}

view indexfile {
  incompatible indexfile;
  IndexFile: readonly;
}
```

Figure 10: Mailpuccino Cache Views

Closer examination revealed that the `MonitoredInputStream` class is not thread safe and its `mark` and `reset` methods are never called. We modified the class to remove these methods and added comments to make it clear that the class is not thread safe.

5.4 TupleSoup

TupleSoup is an open-source database library written in Java. TupleSoup is available at `http://sourceforge.net/projects/tuplesoup/`. TupleSoup contains over 6,600 lines of code. We rewrote all of the synchronization in TupleSoup to use views.

TupleSoup contains three index classes: a `MemoryIndex` class, a `PageIndex` class, and a `FlatIndex` class. The original index classes only permitted one thread to search the index at a time. We created two views per index class: an `access` view and a `modifying` view. Multiple threads can simultaneously hold the `access` view. If one thread holds the `modifying` view of an index, no other thread can hold the `modifying` or `access` views of the index.

The `DualFileTable` class implements a cached table backed by two separate files. The original version of `DualFileTable` contained four separate locks: one lock for each of the two data files, a lock for the cache, and a lock for the statistics counters. We first examined the code to see if we could modify the class to allow multiple simultaneous calls to the `getCacheEntry` cache lookup method. Unfortunately, this method actually mutates a list of least-recently-used cache entries that is used to determine which entries to evict. Therefore, it is not safe to allow multiple

```
view filea {
  incompatible filea;
  fileastream, filearandom,
    fca, fileaposition: readwrite;
  updateRowA(Row row) preferred;
  addRowA(Row row) preferred;
}
view fileb {
  incompatible fileb;
  filebstream, filebrandom, fcb, filebposition:
    readwrite;
  updateRowB(Row row) preferred;
  addRowB(Row row) preferred;
}

view indexcache {
  incompatible indexcache;
  indexcache, indexcacheusage, indexcachefirst,
    indexcachelast: readwrite;
  addCacheEntry(TableIndexEntry entry) preferred;
  updateCacheEntry(TableIndexEntry entry)
    preferred;
  removeCacheEntry(String id) preferred;
  getCacheEntry(String id) preferred;
}

view stat {
  incompatible stat;
  stat_add, stat_update, stat_delete,
    stat_add_size, stat_update_size,
    state_read, stat_read_size,
    stat_cache_hit, stat_cache_miss,
  stat_cache_drop: readwrite;
  readStatistics();
}
```

Figure 11: TupleSoup `DualFileTable` **Views**

threads to simultaneously call the `getCacheEntry` method.

We finally used a straightforward translation to views, shown in Figure 11, which replaces each lock with a corresponding view, and synchronized methods with preferred views for methods. Such a translation is quite straightforward to carry out and enables developers to explicitly express the correlations between fields that the locking structure implicitly encoded. In other words, the views explicitly label the data that each lock protects, and our view compiler provides static assurances that the code never accesses protected fields without holding an appropriate view.

5.5 Discussion

We used the following process for annotating an existing class with views. First, we studied an existing class's locking structure. Next, we proposed a view structure which would protect a related group of fields and methods, typically with a read-only view for accessing state and a read-write view for updating state. We fed this view structure to our compiler, which guaranteed that accesses to protected fields and methods only occur when holding appropriate views.

We found that it was straightforward to replace the traditional Java locking structure with view acquisitions; it sufficed to replace `synchronized(x)` with `acquire(x@v)` and synchronized methods with preferred view methods. Each benchmark took a couple of hours to annotate; the crux was in understanding the existing locking structures.

Our process typically results in an application with increased potential for concurrency. Many of our annotated benchmarks allow multiple threads to simultaneously read state, while ensuring that only one thread can write state.

6. RELATED WORK

We discuss three threads of work related to expressing Java concurrency patterns: type systems which ensure the absence of races; static and dynamic race detection tools; and automatic generation of locking schemes.

Many teams have developed different type systems which ensure that well-typed programs are free of data races. Boyapati, Lee and Rinard have developed type systems which ensure the absence of data races by tracking object ownership [3, 2]. Abadi, Flanagan and Freund have developed RaceFreeJava [7], where developers associate a lock with each shared field and express this information via the type system; the compiler infers additional type annotations and verifies that programs conform to the specified type-based discipline. Bacon, Strom and Tarafdar propose the Guava race-free dialect of Java [1], which forces all members of shared objects to synchronized. Views generalize RaceFreeJava by allowing developers to specify the locking policy for a set of related fields and methods, not just for one field at a time as in the RaceFreeJava case. That is, views allow developers to explicitly express, in one place, the state and methods protected by each lock. Moreover, unlike previous approaches, views are not limited to using simple Java locks to guarantee race-freedom; they can leverage readwrite locks and other more sophisticated approaches to concurrency control. Views provide developers with a flexible mechanism that can be used to implement sophisticated approaches to concurrency control.

An alternate approach to statically ensuring that programs are free of races is to detect these races, either statically or dynamically. The Eraser dynamic race detection tool computes lock sets for memory locations and warns if a memory location is not protected by a lock [15]. Choi et al. have developed a runtime approach that records access events and uses several optimizations to minimize overheads [4]. Marino et al's LiteRace tool uses sampling to minimize overheads [13]. Other dynamic approaches use static analysis to lower the instrumentation overhead [17]. While dynamic race detection is useful, it requires adequate test suites to detect bugs. RacerX instead uses interprocedural static analysis to detect race conditions and deadlocks [6]. Other static analysis include Warlock [16] and Sema [11]. Race detection tools are, in general, useful for detecting bugs in programs. However, they provide developers with little guidance about which fields need to be protected by locks. Any solution requires developers to formulate a suitable concurrency control policy for their system. Views enable developers to express concurrency control policies; the compiler then automatically computes a mechanism for implementing the policy. Views therefore differ from race detection and race-free type systems approaches because those approaches only verify that implemented solutions are free of races.

Another technique related to ours is that of automatically generating locking schemes for critical regions [8, 5, 9]. Typically, such approaches allow developers to specify critical or atomic sections of their programs. Zhang et al. state a minimal lock assignment problem that is similar to the problem of lock synthesis for views, but differs in that it contains information about non-conflicting critical sections that are never executed concurrently and therefore can share locks without limiting concurrency [18]. This body of work must rely on static analysis to generate locks and therefore may generate overly conservative locking schemes. Furthermore, this work does not attempt to detect possible data races arising from accessing shared state outside of critical regions. Views instead start with a data-centric approach: developers declare certain fields (and methods) as belonging to a view, and specify when threads acquire views; the compiler then ensures that the program always acquires appropriate views, and synthesizes a locking strategy which respects the view annotations.

7. CONCLUSION

Views can be an effective tool for implementing sophisticated concurrency control and statically detecting possible concurrency bugs. A developer using views writes a set of view declarations and annotates code with view acquisitions. A view declaration describes which views of an object may be simultaneously held by different threads and the parts of the object interface that the view controls. The partial object interface specifies which fields can be read, which fields can be written, and which methods can be called through the view. Our compiler performs static checks of the view specifications and the program's use of views to detect many concurrency bugs. Our compiler automatically synthesizes a locking scheme that enforces the view compatibility constraints. Our experience indicates that views are simple to program with, support sophisticated fine-grained access control, and can detect concurrency bugs. Our approach promises to ease the difficult task of implementing locking schemes for fine-grained concurrency.

Acknowledgments.

This research was partially supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We would like to thank the anonymous reviewers for their helpful comments and attention to detail, especially with respect to the `resize()` implementation of the `Vector` class.

8. REFERENCES

[1] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[3] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[4] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2002.

[5] M. Emmi, J. S. Fischery, R. Jhala, and R. Majumdar. Lock allocation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[6] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[7] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the International Conference on Programming Language Design and Implementation*, 2000.

[8] R. L. Halpert, C. J. Pickett, and C. Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[9] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *TRANSACT*, 2006.

[10] R. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, 1972.

[11] J. A. Korty. Sema: A lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In *Proceedings of the USENIX Winter Technical Conference*, 1989.

[12] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110, New York, NY, USA, 2009. ACM.

[13] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

[14] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Symposium on Operating Systems Principles*, 1997.

[16] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, 1993.

[17] C. von Praun and T. Gross. Object-race detection. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[18] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2007.