Views: Synthesizing Fine-Grained Concurrency Control

BRIAN DEMSKY
University of California, Irvine
and
PATRICK LAM
University of Waterloo

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.

We present views, a new approach to concurrency control. Views ease the task of implementing sophisticated locking schemes and provide static checks to automatically detect many data races. A view of an object declares a partial interface, consisting of fields and methods, to the object that the view protects. A view also contains an incompatibility declaration, which lists views that may not be simultaneously held by other threads. A set of view annotations specify which code regions hold a view of an object. Our view compiler performs simple static checks which identify many data races. We pair the basic approach with an inference algorithm that can infer view incompatibility specifications for many applications.

We have ported four benchmark applications to use views: portions of Vuze, a BitTorrent client; Mailpuccino, a graphical e-mail client; jphonelite, a VoIP softphone implementation; 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. We have evaluated the performance of a view implementation of a red-black tree and found that views can significantly improve performance over that of the lock-based implementation.

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

Additional Key Words and Phrases: concurrency, language design, static verification

This research was partially supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350.

1. INTRODUCTION

The increasing availability of multi-core processors has prompted a resurgence of interest in parallel software. To work properly, parallel software must use concurrency control mechanisms to ensure that multiple threads of execution do not interfere with each other. Without sufficient concurrency control, race conditions occur, causing undesired and potentially incorrect program behaviors.

The dominant concurrency control mechanism today is the lock. Developers must manually acquire an appropriate lock before accessing a shared resource and release the lock when they are done with the resource. Locks, however, specify implementations, not the underlying policies which motivated the developer's use of locks. The compiler therefore does not get any information about the locking policy, and must simply compile the code as-is. Furthermore, subsequent maintainers of the code must somehow understand the locking policy before implementing changes to the code. Ideally, the policy would be well-documented in comments and kept up-to-date as the code evolves. Our work helps maintainers in both the ideal case, by enabling policies to be explicitly encoded and compiled into locking implementations, and in less-than-ideal cases, by ensuring that the automatically-generated implementation always conforms to the policy.

We therefore introduce the notion of *views*. In our system, developers may specify that certain parts of an object's interface and state (a subset of its fields and methods) are protected by views. A thread may only access a protected part of an object interface after it obtains an appropriate view. Views therefore raise the abstraction level of concurrency control: instead of explicitly acquiring a lock (which may protect anything), the developer requests a view, which documents the parts of program state that it protects. To ensure the absence of races, views which access conflicting parts of program state are declared or inferred to be incompatible. A thread which attempts to obtain a view which is incompatible with some other currently-held view must wait until the view becomes available.

Views have two primary benefits. First, views enable developers to specify concurrency control at a higher level than traditional Java locks do, since views allow developers to enumerate state that needs to be protected (in terms of fields and methods). The compiler can use this higher-level information to implement locking; when appropriate, it can automatically use advanced concurrency primitives like read-write locks. Second, the compiler can detect concurrency control problems using information in the view specifications: it can warn about possible race conditions, unprotected field and method accesses, and view specifications that are likely to be wrong.

1.1 Contributions

We present the following contributions in this paper:

—**View Concept:** We introduce a new concurrency primitive which formulates concurrency control in terms of partial object interfaces, or views. This higher-level abstraction enables developers to specify the underlying concurrency policy, which explicitly identifies the relevant parts of the implementation that need to be protected.

—**Automatic Lock Synthesis:** We present a technique for compiling views into locking primitives. Our technique currently supports both standard Java locks and read-write locks. It uses a greedy algorithm to synthesize implementations of view policies from their specifications.

—**Static Checking:** We describe several static checks for automatically detecting concurrency errors. Our checks identify view specifications that are likely to erroneously allow race conditions, as well as unprotected accesses to data which is protected by a view.

—**Inferring View Incompatibility:** We present an extension that can automatically infer view incompatibility for many applications. This extension analyzes two views' field access descriptions for hazards to infer compatibility.

—**Experience with Views:** We summarize our experience porting four significant benchmarks to use views, and present performance results from microbenchmarks. Our experience indicates that it is relatively simple to use views, that views can support advanced locking primitives, and that views can statically detect potential concurrency bugs.

We have made our compiler (under the GNU General Public License) and benchmark suite publicly available at the following address: `http://demsky.eecs.uci.edu/views/`.

The structure of the remainder of the paper is 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 four 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 an implementation of the `Vector` appropriate for use in single-threaded programs. 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()`, as its implementation is quite similar to that of `resize()`.

Views consist of two parts: view declarations, which identify the members of each view, and view annotations to Java source code, whereby threads acquire views as needed throughout the implementation. Figure 2 presents modifications to lines 28 through 31 of the existing `Vector` code: they add view acquisitions to the code, thereby enabling its safe use in multi-threaded programs. 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.

```
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 }
```

Fig. 1. Sequential Vector Example.

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

Fig. 2. Changes to Vector to support views.

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.

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

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

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

Fig. 3. View Declarations for Vector Example.

2.2 View Declarations

Figure 3 declares five views: `read`, `write`, `xclRead`, `resize` and `capacity`. The `read`, `write` and `capacity` views correspond to methods of `Vector`, and state 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 updates the `Vector`.

View declarations include a view's name and its body. Figure 3 begins with the `read` view. A view body may optionally list views that are incompatible with the current view; two threads may not simultaneously hold incompatible views. Our views compiler can either infer incompatibility declarations or use developer-provided incompatibility declarations. In the 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.

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. Access descriptions for scalar (non-array) fields are one of `none`, `readonly`, or `readwrite`, while access descriptions for array fields may additionally be `arraywrite`, `fieldreadonly`, and `fieldreadwrite`. Line 3 declares that threads holding the `read` view of a `Vector` object may read its `size` and `capacity` fields. Declaring `array` to be `readonly` implies that any thread holding the `read` view may read elements of the array stored at `array`; it may neither write to the array, nor break encapsulation of the `array` in its containing `Vector` object. Line 10 indicates that a thread holding the `write` view has full access to the `size` field and may write to elements of the `array`. Contrast line 10 with line 26, which declares that a thread holding the `resize` view may modify the array reference held in the field `array` (as well as the array elements). Section 3 explains access descriptions, including the array access descriptions, in greater detail.

A method declaration identifies a method as belonging to a view by giving 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.

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 4.3).

2.3 Checking Views

We have implemented an extension to the Polyglot extensible compiler framework [15] 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 each constructor full access to the object under construction. 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` field and reads from the array stored in the `array` field of the `this` object. Both of these fields have readonly access in the `read` view, which permits reads and array accesses. 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 and the `array`'s elements. (Note that `set()` is not allowed to expose the `array` object, which is encapsulated by the `Vector`. Section 4.2 describes how we guarantee encapsulation of arrays.) Additionally, because `set()` calls the `capacity()` method, the compiler checks that the `write` view contains the `capacity()` method. All checks succeed in our example.

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 element `i` of `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 `capacity` and `size` fields and to reassign the value of the `array` field (due to the `arraywrite` access permission). 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

[1]The compiler correctly displays an error message if `resize()` does not belong to any view granting access to `array`.

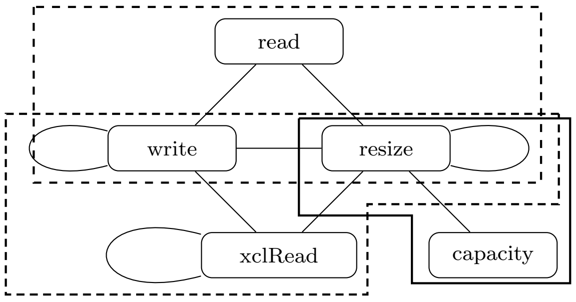

Fig. 4. Incompatibility Graph G for `Vector`.

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] 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 run-time 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.

[2]A read-write lock [13] can be held by any number of threads in read mode but by only one thread in write mode.

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 an optional list of incompatible views followed by a list of view members, which may be fields or methods. The compiler infers incompatibility specifications if the developer omits the incompatibility declaration; an empty list of incompatible views indicates a view which is compatible with all other views. Field members have associated access descriptions. 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.

Access descriptions for scalar, or non-array, fields control access to those fields; holding a view to field `f` with access description `readwrite` permits full access to `f`, while `readonly` permits the holder only to read `f`, and `none` allows no access.

Array fields, however, complicate the picture. Objects often use arrays to store data. Such arrays generally ought to be encapsulated: no references to such arrays should ever become visible (which would permit other parts of the program to have uncontrolled access to array elements). Our view language extensions therefore enable developers to identify encapsulated arrays, and our view compiler ensures that such arrays never escape their containing objects. The implication of this guarantee is that the view system, as a whole, properly controls access to reads and writes of values in encapsulated arrays: the only way to access an encapsulated array is by reading the reference to the array from the containing object's field, and the access only succeeds if the executing thread holds the necessary view. Out of the five access descriptions, the `fieldreadonly` and `fieldreadwrite` descriptions denote unencapsulated arrays, while the usual `readonly` and `readwrite` descriptions specify encapsulated arrays, and `arraywrite` permits mutation of an encapsulated array reference.

We first discuss the descriptions for unencapsulated arrays. Any array `f` which appears in some view with access description `fieldreadonly` or `fieldreadwrite` is treated as if it were a scalar field. The `fieldreadwrite` description permits the holder unlimited read and write access to `f`. The `fieldreadonly` description permits a holder unlimited read access to the field `f`. (Note that, in particular, the holder may expose the reference to field `f`; any recipient of this reference will have complete access to the array elements). Both of these access descriptions permit reads *and writes* to the elements of the array stored in `f`. These access descriptions are incompatible with the other three access descriptions, `readonly`, `readwrite`, and `arraywrite`: no field may be declared as `fieldreadonly` or `fieldreadwrite` in some view and as `readonly`, `readwrite`, or `arraywrite` in any other view.

The three remaining descriptions ensure encapsulation of arrays. The access description `readonly` permits reads from elements of an array `f`, e.g. `o.f[3]`[3]. It

[3]We have changed the semantics of the `readonly` and `readwrite` access description for arrays from our earlier work on views [?]; the `fieldreadonly` and `fieldreadwrite` access descriptions

$$\begin{array}{rcl}
\textit{viewDecl} & := & \texttt{view}\ \textit{name}\ \{\ \textit{incompDecl fieldMethodDecls}\ \} \\
\textit{incompDecl} & := & \varepsilon\ \mid\ \texttt{incompatible}\ \textit{optFieldList}; \\
\textit{optFieldList} & := & \varepsilon\ \mid\ \textit{fieldList} \\
\textit{fieldList} & := & \textit{fieldList},\ \textit{field}\ \mid\ \textit{field} \\
\textit{fieldMethodDecls} & := & \textit{fieldMethodDecls},\ \textit{fieldMethodDecl}\ \mid\ \textit{fieldMethodDecl} \\
\textit{fieldMethodDecl} & := & \textit{fieldDecl}\ \mid\ \textit{methodDecl} \\
\textit{fieldDecl} & := & \textit{fieldList} : \textit{accessDesc}; \\
\textit{accessDesc} & := & \texttt{none}\ \mid\ \texttt{readonly}\ \mid\ \texttt{readwrite} \mid \texttt{arraywrite}\ \mid\ \texttt{fieldreadonly}\ \mid \\
 & & \texttt{fieldreadwrite} \\
\textit{methodDecl} & := & \textit{name}(\textit{formallist})\ \textit{optPreferred}; \\
\textit{optPreferred} & := & \varepsilon\ \mid\ \texttt{preferred}
\end{array}$$

Fig. 5. View Declaration.

$$\begin{array}{rcl}
\textit{viewtype} & := & \textit{typename}@\textit{viewname} \\
\textit{formal} & := & \ldots\ \mid\ \textit{viewtype varname} \\
\textit{varDecl} & := & \ldots\ \mid\ \textit{viewtype varname} \\
\textit{statement} & := & \ldots\ \mid\ \texttt{acquire}(\textit{varname}@\textit{viewname})\ \textit{block}
\end{array}$$

Fig. 6. View Annotations.

does not, however, permit the reference to `f` to escape, e.g. `return o.f`, nor does it permit reassignment of field `f`, e.g. `o.f = a`. Any read of such a field `o.f` must either occur in the context of an array access, `o.f[3]`, or be stored to a fresh local variable, `Object r = o.f`. Values may be read through dereferences of `r`, and `r` may be passed as the source parameter to `System.arraycopy`. However, `r` must not escape: it must not be passed to any other function, nor returned. Similarly, the access description `readwrite` permits both reads and writes of elements of `f`, but maintains prohibitions on copies `w` of the field `f` itself. That is, `w` admits array reads and writes, and may be passed as either the source or target parameter to `System.arraycopy`, but `w` may not escape. To enable mutation and unlimited reads of the array reference, a view must declare field `f` with access description `arraywrite`. Such a description permits statements like `o.f = a`. Section 4.2 presents the compile-time static analysis which we use to ensure that encapsulated arrays do not escape.

4. COMPILING VIEWS

We next describe in detail how we type check views, verify declared encapsulation of arrays, check consistent use of views, and automatically generate a locking strategy

correspond to the earlier semantics of `readonly` and `readwrite`, while the new semantics ensure array encapsulation.

that enforces view incompatibility constraints.

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 initially 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. However, these constraints cannot guarantee encapsulation of arrays, which is required to properly enforce the array behaviors that we allow developers to specify. We therefore chose to design a simple static analysis which can verify that array references are confined to their containing fields, and we describe this analysis below.

4.2 Static Analysis for Array Encapsulation

We verify the array access descriptions from Section 3 using a dataflow analysis. Our analysis prevents references to any array field declared with view `readonly` or `readwrite` from escaping; essentially, it is a variant of escape analysis [3] specialized to our particular application domain. We next describe our analysis, which allows common idioms to safely access arrays stored in fields, yet prohibits the exposure of field references.

Our analysis abstraction associates three flags with each array-typed variable `x` at each program point:

—`NO_ESCAPE`: variable `x` may not escape the current scope;

—`NO_STORE`: variable `x` may not be stored into a field; and

—`NO_WRITE_THRU`: variable `x` admits no writes to array elements.

Each flag records the constraints on variable `x` arising from its past history; for instance, if `x` was read from a view-protected field, then it must not escape, and we mark it with `NO_ESCAPE`.

Our analysis contains two phases, a propagation phase and a verification phase. In the propagation phase, a dataflow analysis computes the constraints which apply to each of the program statements based on their predecessors. There are four

`o.f = x;`	create:	x.NO_ESCAPE, x.NO_STORE
	verify:	!x.NO_STORE (if `f` is `readwrite`)
	verify:	!x.NO_ESCAPE
`Object[] x = o.f;`	create:	x.NO_ESCAPE, x.NO_STORE
	create:	if `f` has only `readonly` access, x.NO_WRITE_THRU
	verify:	at least `readonly` access on f
x escapes (e.g. `return x`)	create:	x.NO_STORE
	verify:	!x.NO_ESCAPE
`x[i] = ...;`	verify:	!x.NO_WRITE_THRU
`System.arraycopy`	verify:	!x.NO_WRITE_THRU on target
`y = x;`	create:	x.NO_STORE, y.NO_STORE
	verify:	!x.NO_ESCAPE

Fig. 7. Rules for ensuring array encapsulation.

sources of constraints for array-typed variables. (1) After a field write `o.f=x`, the referenced value `x` may no longer escape nor be stored to any other field. (2) Field reads `x = o.f` constrain the recipient `x` to no longer be stored nor escape; if the active views of `f` only grant read-only access, then the reference `x` may not be used for writes. (3) Any statement which allows `x` to escape the method scope (e.g. a method call `m(x)` or a return statement `return x`) requires that the method not subsequently store the exposed value `x` to an encapsulated field. (4) After a copy `y = x`, neither `x` nor `y` may not be stored to any encapsulated field.

In the verification phase, the analysis ensures that the program respects the constraints created in the propagation phase, using seven verification rules. At a field write `o.f = x`, it verifies that, if `f` is an encapsulated array field, then `x` has not been marked as ineligible for stores into fields. Furthermore, the field write constitutes an escape, so `x` must not have been marked as non-escaping. Field reads of array references `x = o.f` must read variables `x` which are newly introduced into their scope (`Object[] x = o.f`), to prevent `x` outliving its scope and hence from being accessed without the necessary view on `o`. Furthermore, a field read of field `f` can only succeed with at least `readonly` access to `f`. At any program point which allows `x` to potentially escape the method, e.g. method calls with `x` as a parameter, `x` must have permission to escape. Any write to an element of an array `x` must have write permission for the array. Similarly, any call to `System.arraycopy` requires write permission for the destination array. Finally, for a copy statement `y = x`, we require that variable `x` must have permission to escape. (We could also have required, alternatively, that `y` be prohibited from escaping.)

Figure 7 summarizes the rules for both the propagation and verification phases. The propagation rules appear as "create" clauses, while the verification rules appear as "verify" clauses.

4.3 Static Checks

The compiler also performs several other simple static checks on the view specifications, field accesses, and method calls. These checks help guarantee that view declarations are consistent with each other and that the code does not read or write values to which it does not have access.

—**Read-Write Hazards on Fields and Arrays:** 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. An identical check applies to field writes to and field reads from fields that reference arrays. For each array reference g, the compiler flags the array g if v_1 has array write or write access and v_2 has array write, write, or read access to g. 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(`$\mathtt{a}_1$`, ..., `$\mathtt{a}_N$`)` to method `m(`$\mathtt{f}_1$`, ..., `$\mathtt{f}_N$`)`, the compiler checks that each argument $\mathtt{a}_i$ at the call site matches the view type of the corresponding method formal parameter $\mathtt{f}_i$, if $\mathtt{f}_i$ 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.

4.4 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_j lists v_i 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 [11]. 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.

4.5 Inferring View Incompatibility

While view incompatibility declarations can be used to capture higher-level abstract hazards, they often capture straightforward read-write or write-write hazards on field accesses. Our view compiler can automatically infer view incompatibility declarations in the latter case. The developer indicates that the compiler should infer view incompatibility for a given view by omitting the view's incompatibility declaration. To infer such a view's incompatibility specification, the compiler compares the view with every other view. If another view contains a field access that may conflict with the field accesses in the current view, the inferred specification will declare the views to be incompatible. To detect conflicting field accesses, the compiler uses the same criteria it uses to generate the read-write hazard warnings presented in Section 4.3.

4.6 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 readwrite 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 acquire a view. To avoid the potential for internal deadlocks, the thread cannot hold any of the component locks while blocking. Figure 8 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.

4.7 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 8 does on component locks, but instead uses this strategy on views.

4.8 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

```
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 acquire.
  }
}
```

Fig. 8. Locking Code to Acquire A View.

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; jPhoneLite, a VoIP softphone 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 [15]. 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 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 10 presents the view declarations for the `tracker.BuddyPluginTracker` class. We converted the 5 locks into

```
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;
}
```

Fig. 9. Views for `BuddyPlugin` class.

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.

```
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;
}
```

Fig. 10. Views for `BuddyPluginTracker` class.

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 11 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 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!

Closer examination revealed that the `MonitoredInputStream` class is not thread safe and its `mark` and `reset` methods are never called. We modified the

```
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;
}
```

Fig. 11. Mailpuccino Cache Views

class to remove these methods and added comments to make it clear that the class is not thread safe.

5.4 jPhoneLite

jPhoneLite is an open-source VoIP softphone written in Java. jPhoneLite is available at `http://sourceforge.net/projects/jphonelite/`, and contains 20,000 lines of code. We annotated the core library of version 0.13.1 beta of jPhoneLite.

The class with the most sophisticated locking structure in jPhoneLite is `RTP`, which handles the transmission and reception of real-time content. This class contains two locks, `lockBuffers` and `lockHostPort`, in addition to the built-in lock on `this`.

Figure 0?? presents the views that we created to replace the locks in `RTP` and `RTPChannel`. These views control access to the input buffer, `bufs`, and its helper fields; the destination port information, `remoteip` and `remoteport`; and, in

RTPChannel, the random number generator r. The original code also uses the built-in lock on the RTP object to protect various remaining parts of RTP's state. Because views interoperate smoothly with Java locks, we decided to keep the original lock on RTP.

The readSamples and writeSamples views protect a circular buffer which contains incoming data. While the readSamples view only reads from the buffer bufs, it must update the auxiliary pointers to the buffer, so it has read-write access to bufFull and bufTail. It must therefore be declared as being incompatible with itself. The writeSamples view writes to the buffer bufs as well as other associated state; it is incompatible with both readSamples and writeSamples. Unfortunately, since both of these views write to related state, the compiler must use a traditional Java lock covering all of the state. However, views still enable a developer to describe the locking policy and the anticipated access patterns to the protected state, and our view compiler computes the optimal locking implementation for this case.

The readHostPort and writeHostPort views protect the destination information, which may change during a connection. The RTP class only writes this information, while the related RTPChannel class reads this information. The primary advantage of views in this case is to enable the compiler to ensure that all reads from and writes to the remoteip and remoteport fields are protected by a lock. Our compiler's use of read-write locks could, in principle, lead to preformance improvements in similar usage patterns, but the potential for performance improvements on network access is limited.

5.5 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 threads to simultaneously call the getCacheEntry method.

We finally used a straightforward translation to views, shown in Figure 12, 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

```
public class RTP {
  view readSamples {
    incompatible readSamples;
    bufFull, bufTail: readwrite;
    bufs: readonly;

    getSamples(short data[]) preferred;
  }

  view writeSamples {
    incompatible readSamples, writeSamples;
    bufFull, bufTail: readwrite;
    bufs, bufHead: readwrite;
  }

  view readHostPort {
    incompatible writeHostPort;
    remoteip, remoteport: readonly;
  }

  view writeHostPort {
    incompatible readHostPort, writeHostPort;
    remoteip, remoteport: readwrite;

    change (String remote, int remoteport) preferred;
  }
  // ...
}

public class RTPChannel {
  view rlock {
    incompatible rlock;
    r: readonly;
  }
  // ...
}
```

Fig. 12. Views for `RTP` and `RTPChannel` classes.

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.

5.6 Performance Microbenchmarks

To verify the increased potential for concurrency, we implemented a red-black tree microbenchmark using both views and locks, and tested its performance. We created two thread-safe versions of the `TreeMap` red black tree implementation from

```
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();
}
```

Fig. 13. TupleSoup `DualFileTable` Views

the Java class library. The view version of the `TreeMap` class declares two views: a `read` view and a `write` view. The `read` view is incompatible with the `write` view, and the `write` view is incompatible with both the `read` view and itself. We declared the `get` method as a preferred method for the `read` view and the `put` method a preferred method for the `write` view. The control version of the `TreeMap` class simply declares both the `get` and `put` methods synchronized.

We developed a workload for our thread-safe `TreeMap` implementations. Our workload initializes a `TreeMap` object by adding all integers between 0 and 100,000 to the map in a random order. It then starts a worker thread on 8 cores. Each worker thread performs 1,000,000 operations on the `TreeMap`. The operations are randomly split between reads (calls to `get`) and writes (calls to `put`) and the write percentage is an adjustable parameter. The workload randomly selects keys to either look up or update the mapping.

We executed the microbenchmark on a dual processor quad-core Intel Xeon E5410 2.33 GHz processor with 20 GB of RAM running 64-bit Linux and kernel version

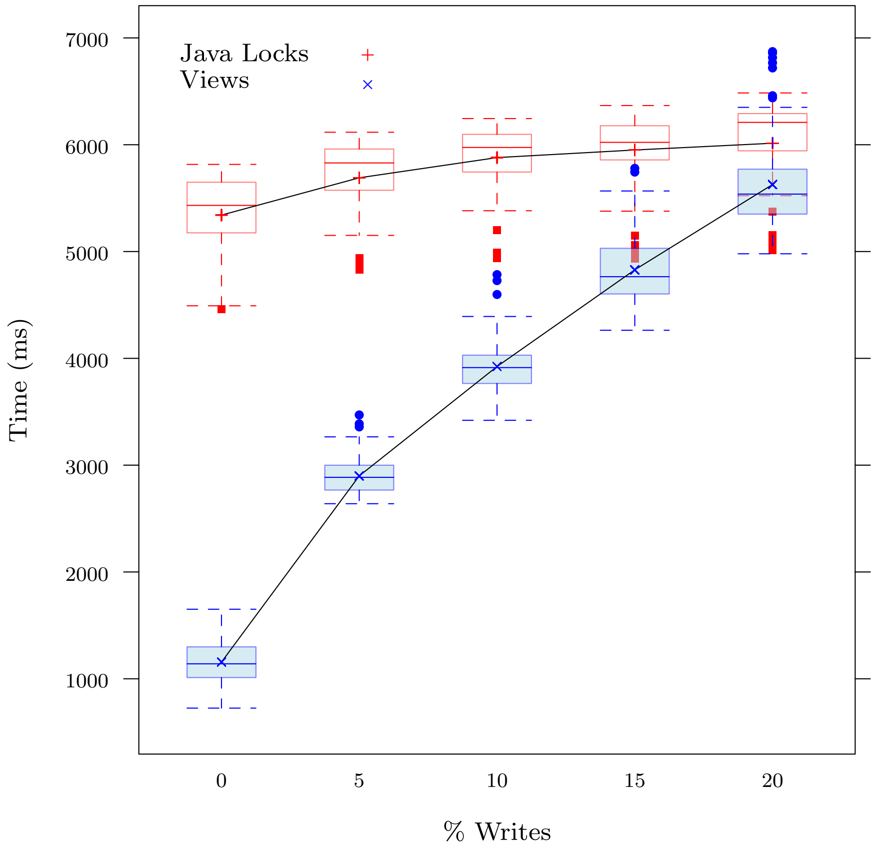

Fig. 14. Time to execute a synthetic workload of red-black tree operations versus percentage of writes in the workload. Lower is better.

2.6.32-rc8. We used workloads with 0%, 5%, 10%, 15%, and 20% writes. We executed each workload on each version 100 times. Figure 13 presents a box plot of the results of this experiment. The bottom and top of each box are the 25th (lower quartile) and 75th (upper quartile) percentiles, respectively, of the distribution. The middle bands are the medians. The crosses are the means. The difference between the lower and upper quartiles is referred to as the interquartile range (IQR). The whisker above the bar is the lowest data point within 1.5 of the IQR of the lower quartile and the whisker below the bar is the highest data point within 1.5 of the IQR of the upper quartile.

The results show that views provide speedups for our microbenchmark of up to 4.6× for read-dominated workloads. We note that this microbenchmark is relatively synchronization heavy — the `get` and `put` methods perform relatively little work compared to the cost of acquiring a lock. Applications that perform more work while holding the locks should see larger speedups. All versions of the benchmark serialize the writes to the red-black tree, so the time to execute the benchmark should increase as the percentage of writes increase. We see this behavior in our experiments with the microbenchmark.

5.7 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.

After we manually developed complete view specifications, we removed the incompatibility declarations and used the incompatibility inference algorithm to automatically infer the incompatibility declarations. We found that the heuristic sucessfully inferred all view incompatibility declarations for Vuze, TupleSoup, and jPhoneLite. The inference algorithm was able to infer incompatibility declarations for all views in Mailpuccino except the `indexfile` view. The `indexfile` view only allows reads from the `IndexFile` field, and therefore it would appear to be safe to allow multiple threads to simultaneously hold this view. However, the view is held while performing file operations on the index file, and serves to protects state changes that occur inside the operating system.

6. RELATED WORK

We discuss three threads of work related to concurrency control. First, we describe systems to prevent or detect races. Next, we discuss other systems for specifying and verifying concurrency control policies in Java. Finally, we compare views to alternate concurrency control mechanisms which go beyond Java lock-based concurrency control.

Ensuring Race-Freedom. Many teams have developed different type systems which ensure that well-typed programs are free of data races. Boyapati, Lee, and Rinard developed type systems which ensure the absence of data races by tracking object ownership [5; 4]. Abadi, Flanagan, and Freund have developed RaceFreeJava [1], 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 [2], 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 read-write 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 alternative 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 [16]. Choi et al. have developed a runtime approach that records access events and uses several optimizations to minimize overheads [6]. Marino et al's LiteRace tool uses sampling to minimize overheads [14]. Other dynamic approaches use static analysis to lower the instrumentation overhead [18]. 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 [8]. Other static analysis approaches include Warlock [17] and Sema [12]. 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 [9; 7; 10]. 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 [19]. 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.

Specifying and verifying design intent. Composable thread coloring [**?**] constrains which threads may execute at specific periods of program executions: one way of avoiding the need to lock shared state is to ensure that it is not shared. For instance, AWT and Swing programs require that only one thread may directly interact with the user interface. We see the work on composable thread coloring as orthogonal to our work on views.

The Fluid project developed techniques for verifying concurrency-related design intent. Part of the work on Fluid [?] enables developers to declare regions and locking policies. Each region controls access to a collection of object fields. It is then the responsibility of the developer to implement a locking policy which satisfies the specification; Fluid can verify locking implementations, as long as they are implemented using standard Java locks, but it cannot implement the locking policy itself. Views compile locking policies into executable code. Because the views compiler understands the underlying policy, it can use advanced concurrency primitives like read-write locks when appropriate.

Other concurrency control mechanisms. Moving beyond Java concurrency, the language-level mechanisms of accept sets and member guards [?] enable developers to implement sophisticated concurrency control mechanisms. Like views, accept sets allow a developer to control access to parts of a method interface. However, accept sets are a dynamic mechanism which temporarily grant and revoke parts of an object's interface. Views differ from accept sets in two main ways. The first difference arises when trying to use an unavailable resource. When using views, it is a compile-time error to attempt to invoke part of the interface that is unavailable (because the thread has not yet acquired the appropriate lock). With accept sets, a caller instead waits until a desired part of an object interface becomes available before invoking that part of the interface. The second difference is in the level of abstraction: views specify which parts of an object need protection, whereas accept sets allow developers to implement a particular locking policy. In particular, accept sets do not explain why these methods are available or not. Because views describe what is being protected, the compiler can identify potential race conditions at compile-time, and the developer can then correct these race conditions before deploying the software.

Member guards enable developers to specify preconditions which must be met before a method may execute. Because member guards can contain arbitrary expressions, they are potentially more expressive than even views; for example, a consumer might only start running when its queue is nonempty. Member guards can therefore express higher-level requirements on methods. Like accept sets, member guards are a dynamic approach which delay method executions until they are safe. However, unlike views, member guards still do not enable developers to state which parts of an object's state need to be protected.

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.

REFERENCES

M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, March 2006.

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.

B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications (OOPSLA)*, pages 20–34, 1999.

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.

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.

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.

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.

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.

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.

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

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

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.

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.

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.

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.

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.

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

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

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.