Static Analyses for Eliminating Unnecessary Synchronization from Java Programs

Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195 USA
{jonal, chambers, egs, eggers}@cs.washington.edu

Abstract. This paper presents and evaluates a set of analyses designed to reduce synchronization overhead in Java programs. Monitor-based synchronization in Java often causes significant overhead, accounting for 5-10% of total execution time in our benchmark applications. To reduce this overhead, programmers often try to eliminate unnecessary lock operations by hand. Such manual optimizations are tedious, error-prone, and often result in poorly structured and less reusable programs. Our approach replaces manual optimizations with static analyses that automatically find and remove unnecessary synchronization from Java programs. These analyses optimize cases where a monitor is entered multiple times by a single thread, where one monitor is nested within another, and where a monitor is accessible by only one thread. A partial implementation of our analyses eliminates up to 70% of synchronization overhead and improves running time by up to 5% for several already hand-optimized benchmarks. Thus, our automated analyses have the potential to significantly improve the performance of Java applications while enabling programmers to design simpler and more reusable multithreaded code.

1. Introduction

Monitors [LR80] are appealing constructs for synchronization because they promote reusable code and present a simple model to the programmer. Many modern programming languages, such as Java [GJS96] and Modula-3, directly support monitors. While these constructs enable programmers to easily write multithreaded programs and reusable components, they can incur significant run time overhead. Reusable code modules may contain synchronization for the most general case of concurrent access, even though particular programs often use these modules in a context that is already protected from concurrency. For instance, a synchronized data structure may be accessed by only one thread at run time, or access to a synchronized data structure may be protected by another monitor in the program. In both cases, unnecessary synchronization increases execution overhead. As described in section 2, even singlethreaded Java programs typically spend 5-10% of their execution time on unnecessary synchronization operations.

Synchronization overhead can be reduced by manually restructuring programs [SNR+97], but this typically involves trading off program performance against simplicity, maintainability, and reusability. To improve performance, synchronization annotations can be omitted where they are not needed for correctness in the current version of the program, or synchronized methods can be modified to provide specialized, fast entry points for threads that already hold a monitor lock. Such specialized functions make the program more complex, and using them safely may require careful reasoning about object-oriented dispatch to ensure that the protecting lock is acquired on all paths to the function call. The assumption that a lock is held at a particular program point may be unintentionally violated by a change in some other part of the program, making program evolution and maintenance error-prone. Hand optimizations make code less reusable, because they make assumptions about synchronization that may not be valid when a component is reused in another setting. In general, complex manual optimizations make programs harder to understand, make program evolution more difficult, reduce the reusability of components, and create an opportunity for subtle concurrency bugs to arise.

In this paper, we present and evaluate static analyses that reduce synchronization overhead by automatically detecting and removing unnecessary synchronization. A synchronization operation is unnecessary if there can be no contention between threads for the synchronization operation. For example, if a monitor is only accessible by a single thread throughout the lifetime of the program, there can be no contention for the monitor, and thus all operations on that monitor can safely be eliminated. Similarly, if threads always acquire one monitor and hold it while acquiring another monitor, there can be no contention for the second monitor, and this unnecessary synchronization can safely be removed. Finally, when a monitor is acquired by the same thread multiple times in a nested fashion, the first monitor acquisition protects the others from contention and therefore all nested synchronization operations may be optimized away. In order to reason statically about synchronization, we assume the compiler has knowledge of the whole program at analysis time; future work may extend our techniques to handle Java's dynamic code loading and reflection features.

There are three main contributions of this paper. First, we describe several synchronization optimization opportunities and measure their frequency of occurrence in several Java programs. Second, we provide precise definitions for a family of analyses designed to detect unnecessary synchronization. Finally, we present a preliminary empirical evaluation of these analyses on a suite of benchmarks. Our partial implementation eliminates up to 70% of synchronization overhead and improves running time by up to 5% for typical Java benchmarks on a highly optimized platform.

The rest of the paper is structured as follows. The next section describes the Java synchronization model, and provides measurements of synchronization overhead for typical benchmarks. Section 3 identifies opportunities for optimizations. Section 4 provides a precise description for a set of analyses that detect and eliminate unnecessary synchronization operations. Section 5 summarizes the performance impact of these analyses on a set of benchmarks, section 6 discusses related work, and section 7 concludes.

2. Java Synchronization

Java provides a monitor construct to protect access to shared data structures in a multithreaded environment.

2.1 Semantics

The semantics of monitors in Java are derived from Mesa [GMS77]. Each object is implicitly associated with a monitor, and any method can be marked `synchronized`. When executing

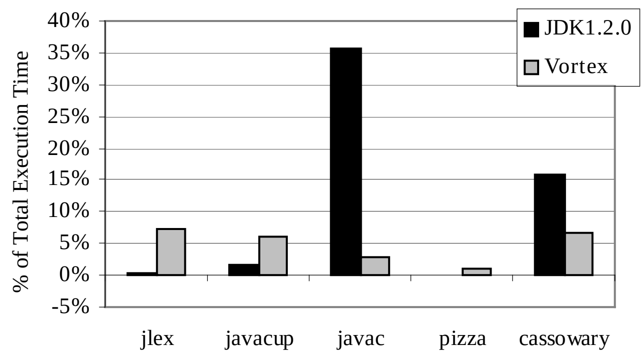

Fig. 1. Overhead of Synchronization

a `synchronized` method, a thread acquires the monitor associated with the receiver object,[1] runs the method's code, and then releases the monitor. An explicit synchronization statement provides a way to manipulate monitors at program points other than method invocations. Java's monitors are reentrant, meaning that a single thread can acquire a monitor multiple times in a nested fashion. A reentrant monitor is only released when the thread exits the outermost method or statement that synchronizes on that monitor.

2.2 Cost

Synchronization represents a significant performance bottleneck for a set of Java benchmarks. To quantify the cost of synchronization operations, we compared singlethreaded Java programs to versions of the same programs where synchronization has been removed from both the application and the standard Java library. Since the correctness of multithreaded benchmarks depends on the presence of synchronization, we did not perform these measurements on multithreaded benchmarks. However, the unnecessary synchronization present in singlethreaded programs suggests that a significant amount of the synchronization in multithreaded programs is also unnecessary.

We used a binary rewriter [SGA+98] to eliminate all synchronization operations from the application binaries. This strategy allowed us to perform measurements on commercial Java virtual machines without having to instrument and recompile them at the source level.

We examine the benchmarks using two different Java implementations that are representative of different Java virtual machine implementations. The JDK 1.2.0 embodies a hybrid JIT compilation and interpretation scheme, and features an efficient implementation of lock operations. Consequently, it represents the state of the art in commercially available Java virtual machines. Vortex, an aggressively optimizing research compiler [DDG+96], produces natively compiled stand-alone executables and uses efficient synchronization primitives [BKM+98]. For these figures, we use the base Vortex system, which does not contain the analyses described in this paper.

Figure 1 shows the percentage of total execution time spent on synchronization in five singlethreaded benchmarks for each platform. Synchronization overhead averages 5-10% of

---

[1] `static synchronized` methods acquire the monitor associated with the `Class` object for the enclosing class.

```
class Reentrant {
   synchronized foo() {
      this.bar()
   }
   synchronized bar()
      { ... }
}
```

```
class Enclosing {
   Enclosed member;
   synchronized foo() {
      member.bar();
   }
}
class Enclosed {
   synchronized bar()
      { ... }
}
```

Fig. 2. Reentrant Monitors

Fig. 3. Enclosed Monitors

execution time, depending on the platform, and can be as high as 35%. The relative cost of synchronization varies between the platforms because of the varying amounts of optimization they perform in the compilation process, and their different synchronization implementations. For example, if Vortex is able to optimize the non-synchronization-related parts of a benchmark like *jlex* more effectively than the JDK 1.2.0, its synchronization overhead will be relatively more significant. In contrast, the benchmarks *javac* and *cassowary* may use synchronization in a way that is more expensive on the JDK platform than on Vortex. Despite the variations between platforms, synchronization overhead represents a significant portion of the execution time for these Java benchmarks, demonstrating that there is considerable room for performance improvement over current synchronization technology.

3. Optimization Opportunities

In this section, we describe three different opportunities for optimizing synchronization operations.

3.1 Reentrant Monitors

Reentrant monitors present the simplest form of unnecessary synchronization. As illustrated in Figure 2, a monitor is reentrant when one synchronized method calls another with the same receiver object. It is safe to remove synchronization from `bar` if all calls to `bar` reachable during program execution are within procedures that synchronize on the same receiver object. Our optimization generalizes this example to arbitrary call paths: synchronization on the receiver object *O* of method `bar` may be removed if along every reachable path in the call graph to `bar` there is a method or statement synchronized on the same object *O*.

If the receiver object's monitor has been entered along some, but not all, call paths to method `bar`, specialization can be used to create two versions of `bar`: an unsynchronized version for the call paths where the receiver is already synchronized, and a synchronized version for the other call paths. The synchronized version acquires the lock and then simply calls the unsynchronized version. For example, if `bar` is also called from the function `main`, where the receiver object is not synchronized, `bar` could be specialized so that `main` calls a synchronized version that acquires a monitor. Methods like `foo` that have already locked the receiver object can still call the more efficient, unsynchronized version of `bar`.

```
class PrintWriter {
   Object lock;
   Writer out;
   void write(int c) {
      synchronized(lock) {
         out.write(c);
      }
   }
}
class StringWriter {
   synchronized write(int c)
        { ... }
}
```

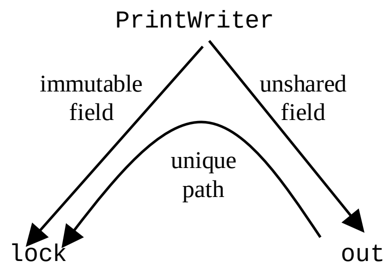

Fig. 4. Immutable Paths

3.2 Enclosed Monitors

An *enclosed monitor* is a monitor that is already protected from concurrent access by another monitor. The enclosing monitor is always entered first, and while it is held the enclosed monitor is acquired. Later, the enclosed monitor and then the enclosing monitor will be released. Because the enclosed monitor is only entered when the enclosing monitor is held, it is protected from concurrent access and is unnecessary. For example, in Figure 3 the monitor on the `member` object is enclosed by the monitor on the `Enclosing` object. Thus the synchronization on the `bar` function is unnecessary and may be removed.

In order to remove synchronization safely from a monitor *M* during static analysis, we must prove there is a unique, unchanging enclosing monitor that protects M, not one of several enclosing monitors. If there were several `Enclosing` objects in Figure 3, for example, different threads could access the `Enclosed` object concurrently by going through different `Enclosing` objects, and it would be unsafe to remove synchronization from `bar`. There are four ways we can ensure this is the case:

First, the enclosing monitor may store the enclosed monitor in an *unshared* field—a field that holds the only reference to the enclosed object. Since the unshared field holds the only reference to the enclosed object, the only way to enter the enclosed object's monitor is to go through the (unique) enclosing object. We can relax the "only reference" condition in the definition of an unshared field if we use the name of the field to identify the enclosing lock. As long each enclosed object is only stored in one instance (i.e., run-time occurrence) of that field, it is permissible for other fields and local variables to refer to the enclosed object, because the field name uniquely identifies the enclosing object.

Second, the enclosing monitor may be stored in an *immutable static field*, i.e. a global variable that does not change value. Because the enclosing monitor is identified by the static field, and only one object is ever stored in that static field, the field name uniquely identifies a monitor. The static field's monitor *M* encloses another monitor *M'* if all synchronization operations on *M'* execute from within monitor *M*.

Third, the enclosing monitor may be stored in an immutable field of the enclosed monitor. Since an immutable field cannot change, the same enclosing monitor is always entered before the enclosed monitor. This case occurs when a method first synchronizes on a field of the receiver object, then on the receiver object itself.

```
class Local {
   synchronized foo()
      { ... }
}
main() {
   new Local().foo();
}
```

Fig. 5. Thread-Local Monitors

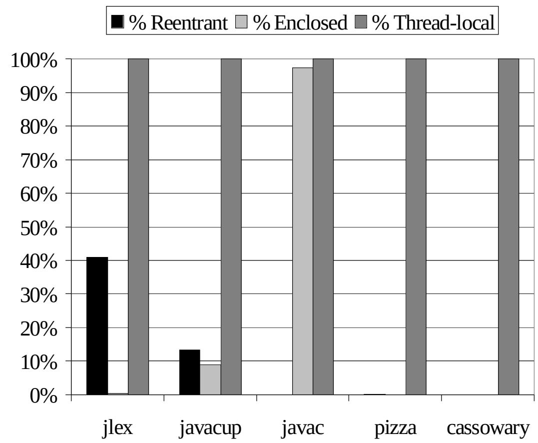

Fig. 6. Optimization Potential

Fourth, the cases above can be combined. For example, Figure 4 illustrates an example similar to cases in the JDK 1.2.0 I/O library when an stream object first synchronizes on an object in one of its fields, then calls a synchronized method on the object in another field. In the example, it is safe to remove the synchronization on `StringWriter.write` because the `lock` object of an enclosing stream is always locked before calling `write`. Since `lock` is an immutable field of `PrintWriter` and `out` is an unshared field of `PrintWriter`, we can use transitivity to determine that there is a unique enclosing object (`lock`) for each enclosed object (`out`). Using transitivity, we can combine a sequence of immutable and unshared fields into a *unique path* from the enclosed monitor to the enclosing monitor. A unique path identifies a unique enclosing object relative to a particular enclosed object.

The general rule we have developed can be stated as follows:

> A synchronization statement *S* may be removed if, for every other synchronization statement *S'* that could synchronize on the same object as *S*, there exists an unique path of links such that:
>
> 1. The first link represents the object synchronized on by *S* and *S'*
> 2. Each subsequent link is either an unshared field of an object that encloses the link before or an immutable field that is enclosed by the link before
> 3. The last link represents an object that is synchronized on all call paths that reach *S* and is also synchronized on all call paths that reach *S'*

As in the case of reentrant monitors, synchronization statements on enclosed objects may be specialized if it is legal to remove synchronization on some instances of a class but not others. For example, the root node in a binary tree encloses all of the inner nodes, so specialization could create two kinds of nodes: one that is synchronized for creating the root of a binary tree, and one that is unsynchronized for creating the inner nodes of the tree.

3.3 Thread-Local Monitors

Figure 5 shows an example of a thread-local monitor. Instances of the `Local` class are only accessible by the thread that created them, because they are created on the stack and are not accessible via any static field. Since static fields are the only base case for sharing data between threads in Java's memory model, it is safe to remove synchronization on methods of any class that is unreachable from static fields. In our model, `Thread` and its subclasses are stored in a global list, so that passing references from one thread to another during thread creation is handled correctly. Specialization can eliminate synchronization when some instances of a class are thread-local and other instances are not.

3.4 Optimization Potential

Figure 6 shows an estimate of the opportunities for optimization in our benchmark suite, demonstrating that different programs present different optimization opportunities. This data was collected from dynamic traces of the five Java programs running on the JDK 1.1.6. For each benchmark, it shows the percentages of dynamic monitor operations that were reentrant, enclosed (by a different monitor), and thread-local, representing an upper bound for how well our analyses could perform. The bars may add up to more than 100% because some synchronization operations may fall into several different categories. All the benchmarks do 100% of their synchronization on thread-local monitors because they are singlethreaded, and so no monitor is ever locked by more than one thread. Multithreaded benchmarks would have some synchronization that is not thread-local, but we believe that thread-local monitors would still represent a significant opportunity in these benchmarks.

The benchmarks differ significantly in the optimization opportunities they present. For example, 41% of the synchronization in *jlex* is reentrant but less than 1% is enclosed. In contrast, 97% of the synchronization in *javac* is enclosed and virtually none is reentrant. For these singlethreaded benchmarks, thread-local monitors present the greatest opportunity for optimization, with two programs gaining significant benefit from enclosing or reentrant monitors. This data demonstrates that each kind of optimization is important for some Java programs.

4. Analyses

We define a simplified analysis language and describe three analyses necessary to optimize the synchronization opportunities discussed above: lock analysis, unshared field analysis, and multithreaded object analysis. Lock analysis computes a description of the monitors held at each synchronization point so that reentrant locks and enclosed locks can be eliminated. Unshared field analysis identifies unshared fields so that lock analysis can safely identify enclosed locks. Finally, multithreaded object analysis identifies which objects may be accessible by more than one thread. This enables the elimination of all synchronization on objects that are not multithreaded. Our analyses can rely on Java's **`final`** annotation to detect immutable fields; an important area of future work is to detect immutable fields that are not explicitly annotated as **`final`**.

4.1 Analysis Language

We describe our analyses in terms of a simple expression-based core language, incorporating the essential synchronization-related aspects of Java. This allows us to focus on the details relevant to specifying the analyses while avoiding some of the complexity of a real language. It is straightforward to handle the missing features of Java—our prototype implementation

```
id, field, fn ∈ ID
label ∈ LABEL
key ∈ KEY
e, program ∈ E
    E ::= new^KEY
     | ID
     | let ID := E1 in E2
     | E.ID
     | E1.ID := E2
     | E1 op E2
     | synchronized^LABEL (E1) { E2 }
     | if E1 then E2 else E3
     | ID(E1,...,En)
```

Fig. 7. Core Analysis Language

handles all of the Java language except reflection and dynamic code loading, which are omitted to enable static reasoning.

Figure 7 presents our analysis language. It is a simple, first-order language, incorporating object creation, field access and assignment, let-bound identifiers, synchronization expressions, and simple control flow. Each object creation point is labeled with a *class key* [GDD+97], which identifies the group of objects created at that point. In our implementation, there is a unique key for each **`new`** statement in the program; in other implementations a key could represent a class, or could represent another form of context sensitivity. We assume that all let-bound identifiers are given unique names. Static field references are modeled as references to a field of the special object `global`, which is implicitly passed to every procedure. We assume all procedures are put into an implicit global table before evaluating the main expression. The *lookup* function returns the λ-expression associated with a particular procedure.

We model ordinary binary operators like `+` and **`;`** (which evaluates and discards its first argument before returning the second) with the E_1 **op** E_2 syntax. Control flow operations include simple function calls and a functional **`if`** expression—facilities that can be combined to form other structures like loops and object-oriented dispatch. Finally, Java's synchronization construct is modeled by a **`synchronized`** statement, which locks the object referred to by E_1 and then evaluates E_2 before releasing the lock. Each **`synchronized`** statement in the program text is associated with a unique label ∈ `LABEL` that is used in our analyses.

4.2 Analysis Context

Our analyses are parameterized by other alias and class analyses, a feature of our approach that allows a tradeoff between analysis time and the precision of our analysis results. Our analyses also benefit from earlier copy propagation and must-alias analysis passes, which merge identifiers that point to the same object. We assume the following functions are defined from earlier analysis passes:

id_aliases(`e`) – the set of identifiers that may point to the same value as expression `e`

field_aliases(`field`) – the set of fields declarations whose instances may point to the same object as `field`. This information can be easily computed from a class analysis.

is_immutable(`field`) – true if `field` is immutable (i.e., write-once). This may be deduced from **`final`** annotations and constructor code.

label_aliases(`label`) – the set of labels of synchronization statements that may lock the same object as the synchronization statement associated with `label`

Some of our analyses deal with groups of objects, represented by class keys. We assume that an earlier class pass has found a conservative approximation to the set of objects that can be in each variable or field in the program. Our implementation uses the 1-1-CFA algorithm [S88][GDD+97], which considers each procedure in one level of calling context and analyzes objects from different creation points separately, and the 0-CFA algorithm, which lacks this context-sensitivity. We use the following functions to access this information:

field_keys(`field`, `key`) – the set of class keys to which field `field` may refer when accessed through a particular class key `key`

static_field_keys(`field`) – the set of class keys to which static field `field` may refer

label_keys(`label`) – the set of class keys that the synchronization expression associated with `label` may lock.

4.3 Analyses

Our analyses compute the following functions:

get_locks(`label`) – the set of locks held at a particular synchronization point denoted by `label`. A lock is represented by a path of two kinds of field links, as described below.

is_unshared(`field`) – true if `field` is unshared

is_multithreaded(`key`) – true if objects described by `key` may be accessible through static variables

We describe our first two analyses in syntax-directed form, where a semantic function maps an expression and a set of inherited attributes to a set of synthesized attributes. The third analysis uses only data from previous analyses and does not work directly over the program text.

Lock Analysis. Figure 8 defines the domains and helper functions that are used by our lock analysis flow functions. Our lock analysis, shown in Figure 9, describes locks in terms of paths and bipaths. A *path* names a particular object relative to an identifier, and consists of the identifier name and a series of field accesses. Thus, the path $\mathtt{id} \rightarrow \mathtt{field}_1 \rightarrow \mathtt{field}_2$ represents the expression $\mathtt{id.field}_1\mathtt{.field}_2$. A *bipath* represents a bi-directional path. The forward links represent field dereferences, as in paths, while the backward links mean "is enclosed by"—that is, in a bipath of the form $bipath_{sub} \leftarrow \mathtt{field}$, the expression denoted by $bipath_{sub}$ is referenced by the `field` field of some unspecified object. In our descriptions, we use the notation $m[x \rightarrow y]$ to denote that we compute a new mapping $\in X \rightarrow Y$ that is identical to mapping m except that element $x \in X$ is mapped to $y \in Y$.

path ∈ `PATH = ID + PATH` → `ID`
dir ∈ `DIR` = { →, ← }
bipath ∈ `BIPATH = ID + BIPATH` × `DIR` × `ID`
lockset ∈ `LOCKSET` = 2^{BIPATH}
lockmap ∈ `LOCKMAP = LABEL` $\rightarrow_{\text{fin}}$ `LOCKSET`
idmap ∈ `IDMAP = ID` $\rightarrow_{\text{fin}}$ 2^{PATH}

is_immutable_path(path) : **bool**
  switch (path)
    case id : **true**
    case path' → field : *is_immutable*(field) ∧ *is_immutable_path*(path')

is_prefix(bipath_1, bipath_2) : **bool**
  if ($\text{bipath}_1 = \text{bipath}_2$) **then true**
  else if (bipath_2 = id) **then false**
  else if (bipath_2 = bipath' dir field) **then** *is_prefix*(bipath_1, bipath')

substitute(bipath_1, path, bipath_2) : `BIPATH`
  if (bipath_1 = path) **then** bipath_2
  else if (bipath_1 = bipath' dir field)
    then *substitute*(bipath', path, bipath_2) dir field
  else error

map_lock(bipath_1, path, bipath_2) : `BIPATH` ∪ { not_defined }
  if (*is_prefix*(path, bipath_1)) **then** *substitute*(bipath_1, path, bipath_2)
  else if (path = path' → field ∧ *is_unshared*(field))
    then *map_lock*(bipath_1, path', bipath_2 ← field)
  else not_defined

map_lockset(lockset, path_1, path_2) : `LOCKSET`
  { *map_lock*(bipath, path_1, path_2) | bipath ∈ lockset } - { not_defined }

Fig. 8. Domains and Helper Functions for Lock Analysis

The lock analysis function *L* accepts four arguments in curried style. The first argument is an expression from the text of the program. The second argument, a lockset, is the set of bipaths representing locks held at this program point. The third argument, a lockmap, is the current mapping from synchronization labels to sets of bipaths representing locks held at each labeled synchronization statement. The final argument, an idmap, is a mapping from identifiers to paths that describe the different field expressions that the identifier aliases. The result of lock analysis is a lockmap that summarizes the locks held at every reachable synchronization label in the program. We analyze the expression representing the program in the context of an empty lockset (no lock encloses the entire program expression), an optimistic lockmap (no synchronization points have been analyzed yet), and an empty idmap (no identifiers are in scope).

Many of the analysis flow functions in Figure 9 are relatively straightforward; we discuss only the more subtle ones below. The rules for **`let`** and `id` expressions update the idmap for identifiers and return the pathset represented by an identifier. A `field` expression simply extends all paths in `e`'s pathset with `field`.

$L : \mathtt{E} \rightarrow \mathtt{LOCKSET} \rightarrow \mathtt{LOCKMAP} \rightarrow \mathtt{IDMAP} \rightarrow 2^{\mathtt{PATH}} \times \mathtt{LOCKMAP}$

get_locks($\mathtt{label}$) : $\mathtt{LOCKSET}$ =
  let (pathset', lockmap') = $L[\![\text{program}]\!]\, \emptyset\ \emptyset\ \emptyset$ **in** lockmap'($\mathtt{label}$)

$L[\![\mathbf{new}^{\mathtt{key}}]\!]$ lockset lockmap idmap = ($\emptyset$, lockmap)

$L[\![\mathtt{id}]\!]$ lockset lockmap idmap = ({ $\mathtt{id}$ } $\cup$ idmap($\mathtt{id}$), lockmap)

$L[\![\mathbf{let}\ \mathtt{id}\ \mathbf{:=}\ \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2]\!]$ lockset lockmap idmap =
  let (pathset', lockmap') = $L[\![\mathtt{e}_1]\!]$ lockset lockmap idmap **in**
    $L[\![\mathtt{e}_2]\!]$ lockset lockmap' idmap[$\mathtt{id} \rightarrow$ pathset']

$L[\![\mathtt{e.field}]\!]$ lockset lockmap idmap =
  let (pathset', lockmap') = $L[\![\mathtt{e}]\!]$ lockset lockmap idmap **in**
    ({ path $\rightarrow$ $\mathtt{field}$ | path $\in$ pathset' }, lockmap')

$L[\![\mathtt{e}_1\mathtt{.field}\ \mathbf{:=}\ \mathtt{e}_2]\!]$ lockset lockmap idmap =
  let (pathset', lockmap') = $L[\![\mathtt{e}_2]\!]$ lockset lockmap idmap **in**
  let (pathset'', lockmap'') = $L[\![\mathtt{e}_1]\!]$ lockset lockmap' idmap **in**
    ($\emptyset$, lockmap'')

$L[\![\mathtt{e}_1\ \mathbf{op}\ \mathtt{e}_2]\!]$ lockset lockmap idmap =
  let (pathset', lockmap') = $L[\![\mathtt{e}_1]\!]$ lockset lockmap idmap **in**
  let (pathset'', lockmap'') = $L[\![\mathtt{e}_2]\!]$ lockset lockmap' idmap **in**
    ($\emptyset$, lockmap'')

$L[\![\mathbf{synchronized}^{\mathtt{label}}\ \mathbf{(}\mathtt{e}_1\mathbf{)}\ \mathbf{\{}\ \mathtt{e}_2\ \mathbf{\}}]\!]$ lockset lockmap idmap =
  let (pathset', lockmap') = $L[\![\mathtt{e}_1]\!]$ lockset lockmap idmap **in**
  let lockmap'' = lockmap'[$\mathtt{label} \rightarrow \bigcup_{path \in pathset'}$ *map_lockset*(lockset, path, $\mathbf{SYNCH}$)] **in**
  $L[\![\mathtt{e}_2]\!]$ (lockset $\cup$ { path | path $\in$ pathset' $\wedge$ *is_immutable_path*(path) }) lockmap'' idmap

$L[\![\mathbf{if}\ \mathtt{e}_1\ \mathbf{then}\ \mathtt{e}_2\ \mathbf{else}\ \mathtt{e}_3]\!]$ lockset lockmap idmap =
  let (pathset', lockmap') = $L[\![\mathtt{e}_1]\!]$ lockset lockmap idmap **in**
  let (pathset'', lockmap'') = $L[\![\mathtt{e}_2]\!]$ lockset lockmap' idmap **in**
  let (pathset''', lockmap''') = $L[\![\mathtt{e}_3]\!]$ lockset lockmap'' idmap **in**
    (pathset'' $\cap$ pathset''', lockmap''')

$L[\![\mathtt{fn}\mathbf{(}\mathtt{e}_1\mathbf{,...,}\mathtt{e}_n\mathbf{)}]\!]$ lockset lockmap_0 idmap =
  let $[\![\lambda\mathbf{(}\mathtt{formal}_1\mathbf{,...,}\mathtt{formal}_n\mathbf{)}\ \mathtt{e}]\!]$ = *lookup*($\mathtt{fn}$) **in**
  $\forall$i $\in$ 1..n **let** (pathset_i, lockmap_i) = $L[\![\mathtt{e}_i]\!]$ lockset lockmap_{i-1} idmap **in**
  let lockset' = $\bigcup_{\substack{i \in 1..n \\ path \in pathset_i}}$ *map_lockset*(lockset, path, $\mathtt{formal}_i$) **in**
  let (pathset', lockmap') = *context_strategy*($L[\![\mathtt{e}]\!]$ lockset' lockmap_n $\emptyset$) **in**
    ({ *substitute*(path, $\mathtt{formal}_i$, path')
      | path $\in$ pathset' $\wedge$ $\exists$i $\in$ 1..n s.t. *is_prefix*($\mathtt{formal}_i$, path) $\wedge$ path' $\in$ pathset_i },
    lockmap')

Fig. 9. Lock Analysis Flow Functions

When a synchronization statement is encountered, the lockmap is updated with all of the bipaths in the lockset. Before being added to the lockset, however, these bipaths are converted to a normal form in terms of e_1, the expression on which the statement synchronizes. This normal form allows us to compare the bipath descriptions of the locks held at different synchronization points in the program in the lock elimination optimization described below.

The normal form expresses a lock in terms of the special identifier **SYNCH** representing e_1, the object being locked. The *map_lockset* function considers each bipath in the lockset in turn and uses *map_lock* to compute a new bipath in terms of a mapping from the pathset of e_1 to **SYNCH**. For each bipath *b* in the lockset, *map_lock* will substitute **SYNCH** into the lock expression bipath if the bipath is a prefix of *b*. For example, if the path corresponding to e_1 is $\mathtt{id} \rightarrow \mathtt{field}_1$ and the lockset is { $\mathtt{id} \rightarrow \mathtt{field}_1 \rightarrow \mathtt{field}_2$ }, then *map_lock*($\mathtt{id} \rightarrow \mathtt{field}_1 \rightarrow \mathtt{field}_2$, $\mathtt{id} \rightarrow \mathtt{field}_1$, **SYNCH**) = **SYNCH** $\rightarrow \mathtt{field}_2$, signifying that the field $\mathtt{field}_2$ of the object referred to by e_1 is already locked at this point.

If the prefix rule does not apply and the last field `field` in the synchronization expression path is unshared, then *map_lock* will try to match against a shorter prefix with **SYNCH** $\leftarrow$ `field` as the expression to be substituted. In Figure 5, the synchronization expression `PrintWriter` $\rightarrow$ `out` is not a prefix of the currently locked object `PrintWriter` $\rightarrow$ `lock`, so since `out` is an unshared field *map_lock* will attempt to substitute **SYNCH** $\leftarrow$ `out` for `PrintWriter` instead. Thus the result we get is *map_lock*(`PrintWriter` $\rightarrow$ `lock`, `PrintWriter` $\rightarrow$ `out`, **SYNCH**) = **SYNCH** $\leftarrow$ `out` $\rightarrow$ `lock`. That is, at the current synchronization point the program holds a lock on the `lock` field of the object whose `out` field points to the object currently being synchronized. This is a correct description of the case in Figure 5.

Next, the expression inside the synchronization block is evaluated in the context of the current lockset combined with all paths in the synchronization expression's pathset that are unique. The *is_immutable_path* function, which checks that each field in a path is immutable, ensures that no lock description is added to the lockset unless it uniquely identifies the locked object in terms of the base identifier.

At function calls, we look up the definition of the called function and evaluate the actual parameters to produce a set of paths for each parameter and an updated lockmap. The *map_lockset* function is used to map actual paths to formal variables in each lock in the lockset. Information about locks that are not related to formal parameters (including the implicit formal parameter `global` mentioned subsection 4.2) cannot be used by the callee, since there would be no way to ensure that the locked object protects any synchronization statements there. The callee is analyzed in the context of the new lockset and lockmap, and the result is memoized to avoid needless recomputation.

Our analysis may be parameterized by a *context_strategy* function that allows a varying level of context sensitivity. The current implementation is context-insensitive—it simply computes the intersections of the incoming lockset with all other locksets and re-evaluates the callee in the context of the new lockset if the input information has changed since the last analysis. We avoid infinite recursion in our analysis by returning an empty pathset and the existing lockmap when a lock analysis flow function is called recursively with identical input analysis information; the analysis will automatically iterate until a sound fixpoint is reached. Since the lockset must decrease in size each time a function is reanalyzed, termination of our analysis is assured. Finally, the set of paths is returned from the function call by mapping back from formals to actuals.

$\mathrm{fset}, \mathrm{shared} \in \texttt{FSET} = 2^{\texttt{ID}}$
$\mathrm{idstate} \in \texttt{IDSTATE} = \texttt{ID} \rightarrow_{\mathrm{fin}} \texttt{FSET}$

$U : \mathsf{E} \rightarrow \texttt{IDSTATE} \rightarrow \texttt{FSET} \rightarrow \texttt{FSET} \times \texttt{IDSTATE} \times \texttt{FSET}$

$\mathit{is\_unshared}(\texttt{field}) = \textbf{let } (\mathrm{idstate}, \mathrm{shared}) = U\llbracket \mathrm{program} \rrbracket \emptyset\ \emptyset \textbf{ in } (\texttt{field} \notin \mathrm{shared})$

$U\llbracket \textbf{new}^{\mathrm{key}} \rrbracket\ \mathrm{idstate}\ \mathrm{shared} = (\emptyset, \mathrm{idstate}, \mathrm{shared})$

$U\llbracket \texttt{id} \rrbracket\ \mathrm{idstate}\ \mathrm{shared} = (\mathrm{idstate}(\mathrm{id}), \mathrm{idstate}, \mathrm{shared})$

$U\llbracket \textbf{let}\ \texttt{id}\ \textbf{:=}\ \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} =$
  $\textbf{let } (\mathrm{fset}', \mathrm{idstate}', \mathrm{shared}') = U\llbracket \mathsf{e}_1 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} \textbf{ in}$
    $U\llbracket \mathsf{e}_2 \rrbracket\ \mathrm{idstate}'[\texttt{id} \rightarrow \mathrm{fset}']\ \mathrm{shared}'$

$U\llbracket \mathsf{e}\texttt{.field} \rrbracket\ \mathrm{idstate}\ \mathrm{shared} =$
  $\textbf{let } (\mathrm{fset}', \mathrm{idstate}', \mathrm{shared}') = U\llbracket \mathsf{e} \rrbracket\ \mathrm{idstate}\ \mathrm{shared} \textbf{ in}$
    $(\mathit{field\_aliases}(\texttt{field}), \mathrm{idstate}', \mathrm{shared}')$

$U\llbracket \mathsf{e}_1\texttt{.field}\ \textbf{:=}\ \mathsf{e}_2 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} =$
  $\textbf{let } (\mathrm{fset}', \mathrm{idstate}', \mathrm{shared}') = U\llbracket \mathsf{e}_1 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} \textbf{ in}$
  $\textbf{let } (\mathrm{fset}'', \mathrm{idstate}'', \mathrm{shared}'') = U\llbracket \mathsf{e}_2 \rrbracket\ \mathrm{idstate}'\ \mathrm{shared}' \textbf{ in}$
  $\textbf{let } \mathrm{fset}''' = \texttt{field} \cup \mathrm{fset}'' \textbf{ in}$
    $(\mathrm{fset}''',$
    $\mathrm{idstate}''[\mathrm{id} \rightarrow \mathrm{idstate}''(\mathrm{id}) \cup \mathrm{fset}''' \mid \mathrm{id} \in \mathit{id\_aliases}(\mathsf{e}_2)],$
    $\textbf{if } (\mathrm{fset}'' \notin \texttt{field}) \textbf{ then } \mathrm{shared}'' \textbf{ else } \mathrm{shared}'' \cup \texttt{field})$

$U\llbracket \mathsf{e}_1\ \textbf{op}\ \mathsf{e}_2 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} =$
  $\textbf{let } (\mathrm{fset}', \mathrm{idstate}', \mathrm{shared}') = U\llbracket \mathsf{e}_1 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} \textbf{ in}$
  $\textbf{let } (\mathrm{fset}'', \mathrm{idstate}'', \mathrm{shared}'') = U\llbracket \mathsf{e}_2 \rrbracket\ \mathrm{idstate}'\ \mathrm{shared}' \textbf{ in}$
    $(\mathrm{fset}'\ \mathrm{merge}_{\mathbf{op}}\ \mathrm{fset}'', \mathrm{idstate}'', \mathrm{shared}'')$

$U\llbracket \textbf{synchronized}^{\mathrm{label}}\ \textbf{(}\mathsf{e}_1\textbf{)}\ \textbf{\{}\ \mathsf{e}_2\ \textbf{\}} \rrbracket\ \mathrm{idstate}\ \mathrm{shared} =$
  $\textbf{let } (\mathrm{fset}', \mathrm{idstate}', \mathrm{shared}') = U\llbracket \mathsf{e}_1 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} \textbf{ in}$
    $U\llbracket \mathsf{e}_2 \rrbracket\ \mathrm{idstate}'\ \mathrm{shared}'$

$U\llbracket \textbf{if}\ \mathsf{e}_1\ \textbf{then}\ \mathsf{e}_2\ \textbf{else}\ \mathsf{e}_3 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} =$
  $\textbf{let } (\mathrm{fset}', \mathrm{idstate}', \mathrm{shared}') = U\llbracket \mathsf{e}_1 \rrbracket\ \mathrm{idstate}\ \mathrm{shared} \textbf{ in}$
  $\textbf{let } (\mathrm{fset}'', \mathrm{idstate}'', \mathrm{shared}'') = U\llbracket \mathsf{e}_2 \rrbracket\ \mathrm{idstate}'\ \mathrm{shared}' \textbf{ in}$
  $\textbf{let } (\mathrm{fset}''', \mathrm{idstate}''', \mathrm{shared}''') = U\llbracket \mathsf{e}_2 \rrbracket\ \mathrm{idstate}'\ \mathrm{shared}' \textbf{ in}$
    $(\mathrm{fset}'' \cup \mathrm{fset}''', \mathrm{idstate}'' \cup \mathrm{idstate}''', \mathrm{shared}'' \cup \mathrm{shared}''')$

$U\llbracket \texttt{fn}\textbf{(}\mathsf{e}_1\textbf{,...,}\mathsf{e}_n\textbf{)} \rrbracket\ \mathrm{idstate}_0\ \mathrm{shared}_0 =$
  $\textbf{let } \llbracket \lambda\textbf{(}\texttt{formal}_1\textbf{,...,}\texttt{formal}_n\textbf{)}\ \mathsf{e} \rrbracket = \mathit{lookup}\,(\texttt{fn}) \textbf{ in}$
  $\forall \mathrm{i} \in 1..\mathrm{n}\ \textbf{let } (\mathrm{fset}_i, \mathrm{idstate}_i, \mathrm{shared}_i) = U\llbracket \mathsf{e}_i \rrbracket\ \mathrm{idstate}_{i-1}\ \mathrm{shared}_{i-1} \textbf{ in}$
  $\textbf{let } \mathrm{idstate}' = \{\ \texttt{formal}_i \rightarrow \mathrm{fset}_i \mid \mathrm{i} \in 1..\mathrm{n}\ \} \textbf{ in}$
  $\textbf{let } (\mathrm{fset}'', \mathrm{idstate}'', \mathrm{shared}'') = \mathit{context\_strategy}(U\llbracket \mathsf{e} \rrbracket\ \mathrm{idstate}'\ \mathrm{shared}_n) \textbf{ in}$
    $(\mathrm{fset}'',$
    $\mathrm{idstate}_n[\mathrm{id} \rightarrow \mathrm{idstate}_n(\mathrm{id}) \cup \mathrm{idstate}''(\texttt{formal}_i) \mid \mathrm{i} \in 1..\mathrm{n} \text{ and } \mathrm{id} \in \mathit{id\_aliases}(\mathsf{e}_i)],$
    $\mathrm{shared}'')$

Fig. 10. Unshared Field Analysis

Unshared Field Analysis. The unshared field analysis described in Figure 10 computes the set of fields that are *shared*, i.e. may refer to objects that are also stored in other instances (that is, run-time occurrences) of the same field. Unshared fields are in the complement of this shared field set. The result of this analysis is used in the *map_lock* function of the previous analysis to detect enclosing locks.

The information computed by unshared field analysis differs from the result of the *field_aliases* function in two essential ways. First, the *field_aliases* function cannot tell whether two instances of a given field declaration may point to the same object, which determines whether a given field is shared. Second, our unshared field analysis is flow-sensitive, enabling increased precision over non-flow-sensitive techniques.

The analysis works by keeping track of which fields each identifier and expression could alias. When a field is assigned a value that may have originated in another instance of the same field, the analysis marks the field shared. *U*, the analysis function for unshared field analysis, accepts as curried parameters a program expression, the set of currently shared fields, and a mapping from identifiers to the sets of fields whose contents they may point to. It then computes the set of fields the expression may alias and an updated set of shared fields. Our analysis is run on the program's top-level expression, using an initially empty identifier mapping (since no identifiers are initially in scope) and initially optimistically assuming that all fields are unshared. The rules for field references, field assignment, and function calls are the most interesting.

When a field `field` is dereferenced, the resulting expression may alias any field in *field_aliases*(`field`). At assignments to a field `field`, we must update the identifier mapping for any identifier that could alias the expression being assigned to `field`, since the values these identifiers point to could also be referenced by `field` due to the assignment. In fact, due to the actions of other threads, these identifiers could alias any field in *field_aliases*(`field`). For the purposes identifying unshared fields, however, we can optimistically assume that such aliasing does not occur when writing to a field. This enables our analysis to detect unshared fields even when the same object is written to two fields with different names. If this object is later copied from one field to another, the field written to will be correctly identified as shared because aliasing is accounted for when reading fields. If the expression being assigned may not alias the field being assigned, then the field being assigned may remain unshared; otherwise, it is added to the shared set. In expressions of the form e_1 **op** e_2, the correct merge function for the expression's field set depends on the operator. For example, the merge function for the **;** operand simply returns the field set of its second argument.

At a function call, we lookup the callee and evaluate all the argument expressions to get a set of fields for each of them as well as an updated identifier map and shared field set. The idstate for the callee consists of a mapping from its formal parameters to the field sets of each actual parameter expression. We then evaluate the callee in the context of the new idstate and the current shared set, and return the resulting field set and shared set. After evaluating the callee, it is also necessary to update the identifier state of the caller. Every id that may alias an actual expression could now reference any field that the formal parameter of the callee could reference after evaluating the callee. This update is necessary because some callee (possibly several levels down the call graph) may have assigned the parameter's value to a field.

Our *context_strategy* for this analysis is context sensitive, as we re-evaluate the callee for each different identifier mapping. In practice, context sensitivity enables results that are much more precise. For example, when a callee is called with a formal parameter aliased to `field` at one call site, we don't want all other call sites to see that the formal may alias `field` after the call and thus conservatively assume that the callee assigned that formal to `field`. Termination is assured because the results of each analysis are memoized, and the size of the field sets is bounded by the number of fields in the program. Recursive functions are handled

$$multi = \left(\bigcup_{f \in staticFields} static\_field\_keys(f) \right) \bigcup \left(\bigcup_{\substack{k \in multi \\ f \in fields(k)}} field\_keys(f, k) \right)$$

Fig. 11. Multithreaded Object Analysis

by optimistically returning the empty set of fields at recursive calls, and the analyses subsequently iterate until a sound fixpoint is reached.

Multithreaded Object Analysis. We define *multi*, a set of class keys, as the smallest set satisfying the recursive equation shown in Figure 11. Then we define *is_multithreaded* as follows:

is_multithreaded(`key`) = `key` $\in$ *multi*

Our implementation simply starts with class keys referenced by static fields, and for each class key it considers each field of that key and adds the keys that field may reference to the set *multi*. When the set reaches the least fixed point, the analysis is complete. The analysis must terminate because there is a finite number of class keys to consider.

4.4 Applying the Results

To apply the results of our analyses, we perform an optimization pass during code generation. At each statement of the form

`synchronized`$^{\texttt{label}}$ **`(`**$\texttt{e}_1$**`) {`** $\texttt{e}_2$ **`}`**

we replace it with the statement $\texttt{e}_1$**`;`** $\texttt{e}_2$ if any of the following conditions holds:

1. **SYNCH** $\in$ *get_locks*(label), or
2. $\forall$ `label'` $\in$ *label_aliases*(`label`) .
   get_locks(`label`) $\cap$ *get_locks*(`label'`) $\neq \emptyset$, or
3. $\forall$ `key` $\in$ label_keys(`label`) . $\neg$is_multithreaded(`key`)

The first condition represents a reentrant monitor—if the monitor associated with expression $\texttt{e}_1$ is already locked, then **`SYNCH`** $\in$ *get_locks*(`label`). Here *get_locks*(`label`) is defined (in Figure 9) to be the result of lock analysis at the program point identified by `label`. We can safely replace the synchronized expression with a sequence that evaluates the lock expression (for potential side effects) and then evaluates and returns the expression protected within the synchronization statement. The second condition represents the generalization to enclosed locks: a synchronization statement *S* may be eliminated if, for every other synchronization statement *S'* that may lock the same object, some common lock is already held at both *S* and *S'*. The third condition removes synchronization statements that synchronize on an expression that refers only to non-multithreaded class keys.

Due to the complicated semantics of monitors in Java, our optimizations may not conform to the Java specification on some multiprocessor systems. According to the Java language specification, "locking any lock conceptually flushes all variables from a thread's working memory, and unlocking any lock forces the writing out to main memory of all variables that the thread has assigned." [GJS96] This implies, for example, that a legal Java program may pass data (in a timing-dependent manner) from one thread to another by having each thread synchronize on a thread-local object. This kind of "covert channel" communication could be broken by our optimizations. An implementation that synchronizes the caches of a multiprocessor even when other parts of a synchronization operation have been removed would comply with the Java specification, for example. Our optimizations are always safe, however, in a Java-like language with a somewhat looser synchronization guarantee which could be informally stated as follows: if thread T_1 writes to a variable V and then unlocks a lock and thread T_2 locks the same lock and reads variable V, then thread T_2 will read the value that T_1 wrote. We believe that most well written multithreaded programs in Java use this model of synchronization.

5. Results

A preliminary performance evaluation shows that a subset of our analysis is able to eliminate 30-70% of the synchronization overhead in several of our benchmarks. We have implemented prototype versions of reentrant lock analysis and multithreaded object analysis, and transformations that use the results of these analyses. Our implementation does not yet apply specialization to optimize different instances of an object or method separately. Although our results are preliminary, they demonstrate the promise of our approach. We plan to complete and evaluate a more robust and detailed implementation in the future, which will include unshared field analysis and enclosed lock analysis.

We demonstrate the performance benefit of our analyses on the five singlethreaded benchmarks presented earlier. While these benchmarks could be optimized trivially by removing all synchronization, they are real programs and may be partly representative of how synchronization is used in multithreaded programs as well. *Javac*, *javacup*, *jlex*, and *pizza* are all compiler tools; *cassowary* is a constraint solver. We hope to evaluate our techniques on multithreaded programs in the future.

Our prototype implementation is built on the Vortex compiler infrastructure [DDG+96] augmented with a simple, portable, non-preemptive, user-level threading package based on QuickThreads [K93]. We compiled all programs with a full suite of conventional optimizations, as well as interprocedural class analysis. For our small benchmarks, we used a 1-1-CFA call graph construction algorithm [GDD+97]; this did not scale well to *pizza*, *javac*, and *javacup*, so we used a simpler 0-CFA analysis for these programs, possibly missing some optimization opportunities due to more conservative alias information. Our lock implementation is already highly optimized, using an efficient lock implementation [BKM+98]. We compiled two versions—one with and one without our synchronization optimizations. Both versions included all other Vortex optimizations. All our runtime overhead measurements come from the average of five runs on a SPARC ULTRA 2 machine with 512 MB of memory. We ran the benchmarked program once before the data were collected to eliminate cold cache startup effects.

Table 1 shows statistics about how our analyses performed. The first two columns show the total number of classes in the program and the number identified as thread-local. Multithreaded object analysis identified a large fraction of classes as singlethreaded for the *jlex*, *javacup*, and *cassowary* benchmarks, but was less successful for *javac* or *pizza*. Since these benchmarks are singlethreaded, all their classes are thread-local. However, because our analysis assumes static field references make a class reachable by other threads, our analysis is only able to determine this for a subset of the classes in each program.

Table 1. Synchronization Analysis Statistics

Benchmark	classes		total lock ops	lock ops removed			% overhead removed
	total	thread-local		reentrant	thread-local	total	
jlex	56	34	27	2	8	9	67%
pizza	184	0	38	6	0	6	N/A
javacup	66	28	30	2	6	7	47%
cassowary	57	29	32	4	12	13	27%
javac	194	0	68	5	0	5	0%

The next four columns of Table 1 show the total (static) number of synchronization operations, the number removed by reentrant lock analysis, the number of thread-local operations removed, and the total number of operations removed. The total is less than the sum from the two analyses because some synchronization operations were removed by both analyses. As suggested by the class figures in the first two columns, multithreaded object analysis was more effective than reentrant lock analysis for *jlex*, *javacup*, and *cassowary*, while *pizza* and *javac* only benefited from reentrant lock analysis. In general, our analyses removed 20-40% of the static synchronization operations in the program.

The last column summarizes our runtime performance results. We present the speedup achieved by our optimizations as a percentage of the overhead of synchronization for Vortex. For *jlex*, *javacup*, and *cassowary*, we eliminated a significant percentage of the synchronization overhead, approaching 70% in the case of *jlex*. The absolute speedups ranged up to 5% in the case of *jlex*. Pizza did not have a significant overhead from synchronization, so no speedup was achievable. We also got no measurable speedup on *javac*.

The speedup in the case of *jlex* is due the large number of stack operations performed by this benchmark, which our analysis optimized effectively. Multithreaded analysis discovered that all of the `Stack` objects were thread-local, and lock analysis was successful in removing some reentrant locks in the `Stack` code. Most of the remaining synchronization is on `DataOutputStream` and `BufferedOutputStream` objects. Multithreaded object analysis determined that `DataOutputStream` was thread-local and that the most important instances of `BufferedOutputStream` were thread-local, but because our implementation does not yet produce specialized code for instances of `BufferedOutputStream` that are thread-local we were unable to take advantage of this knowledge. Implementing specialization would improve our optimization performance here.

Over 99% of *Javacup*'s synchronization comes from manipulation of strings, bitsets, stacks, hashtables, and I/O streams. Multithreaded analysis was able to remove synchronization from every method of `StringBuffer`, but was did not eliminate synchronization from other objects. Each of the other classes was reachable from a static variable, either in the Java library or in the *javacup* application code.

To optimize this code effectively would require three additional elements. First, we need a scalable analysis that distinguishes program creation points so that one multithreaded `Hashtable` does not make all `Hashtable`s multithreaded. Our current 1-1-CFA analysis that distinguishes creation points does not scale to *javacup* or *javac*, and therefore our performance suffers for both benchmarks. Second, we need specialization to optimize different instances of the same class separately. Third, we need a more effective multithreaded analysis that can determine if a static variable is only used by one thread, rather than conservatively assuming all such variables are multithreaded.

In *Cassowary*, multithreaded analysis was able to remove synchronization from all the methods of `Vector`. However, the primary source of synchronization overhead was `Hashtable`, which was not optimized by our multithreaded analysis because it was reachable from static fields.

Although a few operations were optimized in *javac*, we did not measure any speedup in this benchmark. Since *javac* executes many operations on enclosed monitors, we expect these results to improve once we have implemented our unshared field analysis and enclosed lock analysis.

Considering that we have achieved a significant fraction of the potential speedup for several of our benchmarks although many important elements of our analyses are not yet implemented, we find these results promising.

6. Related Work

A large body of work (e.g., [ALL89] [KP98]) has focused on reducing the overhead of locking or synchronization operations. Most recently, Bacon's Thin Locks [BKM+98] reduce the overhead of Java's synchronization to a few instructions in the common case. Thin locks improve the performance of real programs by up to 70% by reducing the latency of individual synchronization operations. Our analyses complement this work by reducing the number of synchronization operations.

Diniz and Rinard [DR98] present two techniques for lock coarsening in parallelizing compilers: merging multiple locks into one, so that several objects are protected by one lock, and transforming locks that are repeatedly acquired and released within a method so that they are only acquired and released once. Their work is applicable to explicitly parallel programs; however, they do not evaluate their optimizations in this context. They do not consider thread-local locks, do not consider immutable fields as a potential source of lock nesting, and apparently can only optimize nested locks in languages like C++ where objects can be statically declared to be represented inline. Their coarsening optimizations are complementary to our work; while we can eliminate a broader class of redundant locks, their optimizations may lead to acquiring the non-redundant locks fewer times.

Another source of related work is the Concert project at the University of Illinois. To reduce the overhead of lock operations, they optimize calls from one method to another on the same receiver by eliminating the lock operation from the second method during inlining [PZC95]. They also do a lock coarsening optimization similar to that in [DR98]. Our research extends and generalizes their results by optimizing enclosing locks and thread-local objects.

Our concept of an unshared field is similar to idea of a unique pointer [M96] or unique aliasing mode [H91][NVP98]. Unlike the previous work, we find unshared fields automatically, rather than requiring annotations from the programmer. Our unshared field analysis is similar to an analysis used by Dolby to inline object fields [D97]. In order to safely inline a field, his system propagates tags to determine which fields could alias particular variables. The precision of his analysis is identical to ours given a similar analysis framework, but his work requires more strict conditions to inline a field than ours requires to identify an unshared field.

Work from the model-checking community [C98] performs shape analyses similar to ours in order to simplify models of concurrent systems. These analyses remove recursive locks and locks on thread-local objects from formal models. This allows a model checker to more easily reason about the concurrency properties of a Java program. An analysis similar to enclosing lock analysis is also performed, not to eliminate enclosed locks, but to reason about which objects might be subject to concurrent (unprotected) access. The analyses are intraprocedural, and thus are only applicable to small programs where all methods are inlined. The work does not describe the analyses precisely, nor does it consider the potential performance improvements of removing unnecessary synchronization. Our work precisely describes a family of interprocedural analyses for removing unnecessary synchronization and provides an initial evaluation of their effects on set of benchmarks.

The Extended Static Checking System [DRL+98] allows a programmer to specify a locking protocol using code annotations. The program is then checked for simple errors such as deadlocks or race conditions. This system complements ours by focusing on the correctness of the source code, while our analyses increase the efficiency of the generated code.

7. Conclusion

This paper presented a set of interprocedural static analyses that effectively detect and eliminate unnecessary synchronization. These analyses identify excess synchronization operations due to reentrant locks, enclosed locks, and thread-local locks. A partial implementation of our analyses eliminates 30-70% of synchronization overhead on three Java benchmarks. Our optimizations support a style of programming in which synchronization code is written for software engineering objectives rather than hand-optimized for efficiency.

Acknowledgements

This work has been supported in part by a National Defense Science and Engineering Graduate Fellowship from the Department of Defense, NSF grant CCR-9503741, NSF Young Investigator Award CCR-9457767, and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software. We appreciate feedback and pointers to related work from David Grove, Martin Rinard, William Chan, Satoshi Matsuoka, members of the Vortex group, and the anonymous reviewers. We also thank the authors of our benchmarks: JavaSoft (*javac*), Philip Wadler (*pizza*), Andrew Appel (*jlex* and *javacup*), and Greg Badros (*cassowary*).

References

[ALL89] T. E. Anderson, E. D. Lazowska and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. IEEE Transactions on Computers 38(12), December 1989, pp. 1631-1644.

[BKM+98] D. Bacon, R. Konuru, C. Murthy, M. Serrano. Thin Locks: Featherweight Synchronization for Java. In Proceedings of the 1998 Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998.

[BW88] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. Software Practice & Experience, September 1988, pp. 807-820.

[C98] J. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. In Proceedings of the International Symposium on Software Testing and Analysis, March 1998. A more recent version is University of Hawaii ICS-TR-98-20, available at http://www.ics.hawaii.edu/~corbett/pubs.html.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In Proceedings of the Eleventh

Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 1996.

[DRL+98] David L. Detlefs, K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Compaq SRC Research Report 159. 1998.

[DR98] P. Diniz and M. Rinard. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-based Programs. In Journal of Parallel and Distributed Computing, Volume 49, Number 2, March 1998, pp. 218-244.

[D97] J. Dolby. Automatic Inline Allocation of Objects. In Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1997.

[GMS77] C. M. Geschke, J. H. Morris and E. H. Satterthwaite. Early Experiences with Mesa. Communications of the Association for Computing Machinery, 20(8), August 1977, pp. 540-553.

[GJS96] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[GDD+97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. In Proceedings of the 12th Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1997.

[H91] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In Proceedings of the Sixth Conference on Object-Oriented Programming, Systems, Languages, and Applications, November 1991.

[K93] D. Keppel. Tools and Techniques for Building Fast Portable Thread Packages. University of Washington Technical Report UW CSE 93-05-06, May 1993.

[KP98] A. Krall and M. Probst. Monitors and Exceptions: How to implement Java efficiently. ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

[LR80] B. Lampson and D. Redell. Experience with Processes and Monitors in Mesa. In Communications of the Association for Computing Machinery 23(2), February 1980, pp. 105-117.

[M96] N. Minsky. Towards Alias-Free Pointers. In Proceedings of the 10th European Conference on Object Oriented Programming, Linz, Austria July 1996.

[NVP98] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In Proceedings of the 12th European Conference on Object Oriented Programming, Brussels, Belgium, July 1998.

[PZC95] J. Plevyak, X. Zhang, and A. Chien. Obtaining Sequential Efficiency for Concurrent Object-Oriented Languages. In Proceedings of the 22nd Symposium on Principles of Programming Languages, San Francisco, CA, January 1995.

[S88] Olin Shivers. Control-Flow Analysis in Scheme. SIGPLAN Notices, 23(7):164-174, July 1988. In Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.

[SNR+97] S. Singhal, B. Nguyen, R. Redpath, M. Fraenkel, and J. Nguyen. Building High-Performance Applications and Services in Java: An Experiential Study. IBM T.J. Watson Research Center white paper, available at http://www.ibm.com/java/education/javahipr.html. 1997.

[SGA+98] E. G. Sirer, A. J. Gregory, N.R. Anderson, B.N. Bershad. Distributed Virtual Machines: A System Architecture for Network Computing. In Proceedings of the Eighth ACM SIGOPS European Workshop, September 1998.