# Comprehensive Synchronization Elimination for Java

Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, and Susan J. Eggers

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle WA 98195-2350
{jonal,egs,chambers,eggers}@cs.washington.edu

**Abstract**

In this paper, we describe three novel analyses for eliminating unnecessary synchronization that remove over 70% of dynamic synchronization operations on the majority of our 15 benchmarks and improve the bottom-line performance of three by 37-53%. Our analyses attack three frequent forms of unnecessary synchronization: thread-local synchronization, reentrant synchronization, and enclosed lock synchronization. We motivate the design of our analyses with a study of the kinds of unnecessary synchronization found in a suite of single- and multithreaded benchmarks of different sizes and drawn from a variety of domains. We analyze the performance of our optimizations in terms of dynamic operations removed and run-time speedup. We also show that our analyses may enable the use of simpler synchronization models than the model found in Java, at little or no additional cost in execution time. The synchronization optimizations we describe enable programmers to design efficient, reusable and maintainable libraries and systems in Java without cumbersome manual code restructuring.

## 1. Introduction

Monitors [LR80] are appealing constructs for synchronization, because they promote reusable code and present a simple model to the programmer. For these reasons, several programming languages, such as Java [GJS96] and Modula-3 [H92], directly support them. However, widespread use of monitors can incur significant run-time overhead: reusable code modules such as classes in the Java standard library often contain monitor-based synchronization for the most general case of concurrent access, even though particular programs use them in a context that is already protected from concurrency [HN99]. 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 single-threaded Java programs typically spend 10-50% of their execution time on synchronization operations.

Synchronization overhead can be reduced by *manually* restructuring programs [SNR+97], but any performance improvement gained typically comes at the cost of simplicity, maintainability, reusability, and even program correctness. For example, 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 to ensure that the protecting lock is acquired on all paths to the function call. In addition, 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. A second restructuring technique removes synchronization annotations where they are not needed for correctness in the current version of the program. Both of these 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. These assumptions create an opportunity for subtle concurrency bugs to arise over the course of program evolution. Overall, complex manual optimizations make programs harder to understand, make program evolution more difficult, reduce the reusability of components, and can lead to subtle concurrency bugs.

In this paper, we present and evaluate static analyses that reduce synchronization overhead by *automatically* detecting and removing unnecessary synchronization. The analyses eliminate synchronization from code that can only be executed by a single thread, synchronization on locks already protected by an enclosing lock, and synchronization on reentrant locks. The analyses provide several advantages over manual optimization. First, because our analyses are run automatically during compilation, the source code is left in its original form, thus avoiding the code complexity and error-prone program evolution that results from manual restructuring. Second, automatic analyses avoid the significant effort involved in manual restructuring. Third, our analyses may make other static analyses (e.g., model checking [C98]) more tractable by reducing the number of concurrency constructs in the program.

Finally, our analyses allow programmers to use a more general language model in which every public method synchronizes on the receiver's monitor, rather than ad-hoc synchronization on some methods and not others. Present in several concurrent object-oriented languages [P96], this model considerably simplifies programmer reasoning about race conditions by moving locking granularity to the level of the class. The programmer can rely on the compiler to remove extra synchronization statements in particular contexts, thus ensuring safe multithreaded interaction, while at the same time avoiding a large run-time performance penalty. In general, this technique could lead to deadlock and reduced concurrency, so it would be desirable to provide a way to override the default synchronization. However, if a program deadlocks, the problem is often easily identified by looking at which threads hold which locks; data corruption due to race conditions caused by manual optimization may be much more difficult to debug, because it is usually detected long after the race condition occurs.

To evaluate our analyses, we performed two sets of experiments on a set of single- and multithreaded Java applications in which programmers had manually optimized synchronization. We analyzed the applications to show the extent to which our analyses could further improve upon programmer efforts to eliminate synchronization operations. The thread-local analysis was particularly important here: it was responsible for eliminating the majority of synchronization and dramatically outperformed previously published analyses. Overall, our analyses removed a mean of 88% of dynamic synchronization operations for singlethreaded benchmarks and 35% for multithreaded benchmarks, with a high of over 99%. The effect on execution times for three of the benchmarks was an increase in performance of over 37%; other benchmarks we evaluated also improved, but to a far lesser extent, because the dynamic frequency of synchronization operations in them was low. Our results demonstrate that automatically detecting and removing synchronization overhead can eliminate the drawbacks of manual removal, while still improving application performance.

In addition, to demonstrate that our analyses would allow a more programmer-friendly synchronization model, we simulated the effect of this model by adding concurrency to all public methods of our benchmarks. The results show that our algorithms are able to remove virtually all of the overhead associated with such a model. In the past, this model of concurrency had been regarded as too expensive, since it may lead to much more synchronization than in more conventional models.

This paper makes four contributions. First, we empirically evaluate the types of unnecessary synchronization in a wide range of single- and multi-threaded benchmarks, demonstrating the potential benefit of several types of optimization. Second, we provide a formal presentation of precise and efficient algorithms for detecting three kinds of unnecessary synchronization. Third, we evaluate the performance of our algorithms on the same set of applications, and analyze dynamic synchronization behavior, the contribution of the individual analyses to overall performance, and the benefits of our analyses relative to previous studies. Finally, we demonstrate that our analyses make a simpler model of synchronization feasible, by effectively removing synchronization overhead from a simple motivating example program, as well as our original multi-threaded benchmarks with synchronization added to all public methods.

The rest of the paper is structured as follows. The next section briefly describes the Java synchronization model, and motivates our research with measurements of synchronization overhead in both single- and multithreaded benchmarks. Section 3 compares our analyses to several recently published algorithms that also strive to eliminate synchronization in Java. Section 4 describes our thread-local, reentrant lock, and enclosed lock analyses. Section 5 presents our performance results. Finally, section 7 concludes.

## 2. The Synchronization Problem

### 2.1 Cost of Synchronization in Java

Synchronization is expensive in Java programs, typically accounting for a significant fraction of execution time. Although it is difficult to measure the cost of synchronization directly, it can be estimated in a number of ways. Microbenchmarks show that individual synchronization operations take between 0.14 and 0.4 microseconds even for efficient synchronization implementations running on 400MHz processors [BH99][KP98]. Our own whole-program measurements show that a 10-40% overhead is typical for single-threaded[1] applications in the JDK 1.2.0. Another study found that several programs spend between 26% and 60% of their time doing synchronization in the Marmot research compiler [FKR98]. Because synchronization consumes a large fraction of many Java programs' execution time, there is a potential for significant performance improvements by optimizing it away.

### 2.2 Types of Unnecessary Synchronization

A synchronization operation is *unnecessary* if there can be no contention between threads for its lock. In order to guide our synchronization optimizations, we have identified three important classes of unnecessary synchronization that can be removed by automatic compiler analyses. First, if a lock is only accessible by a single thread throughout the lifetime of the program, *i.e.*, it is *thread-local*, there can be no contention for it, and thus all operations on it can safely be eliminated. Similarly, if threads always acquire one lock and hold it while acquiring another, *i.e.*, the second lock is *enclosed*, there can be no contention for the second lock, and the synchronization operations on it can safely be removed. Finally, when a lock is acquired by the same thread multiple times in a nested fashion, *i.e.*, it is a *reentrant* lock, the first lock acquisition protects the others from contention, and therefore all nested synchronization operations can be optimized away.

It is possible to imagine other types of unnecessary synchronization, such as locks that protect immutable data structures, locks that do not experience contention due to synchronization mechanisms other than enclosing locks, and acquiring and releasing a lock multiple times in succession [DR98]. We focus on the three types discussed above, because they represent a large proportion of all unnecessary synchronization, they can be effectively identified and optimized, and their removal does not impact the concurrency behavior of the application. We define two analyses to optimize these types of unnecessary synchronization: *thread-local analysis* to identify thread-local locks, and *lock analysis* to find enclosed locks and reentrant locks.

[1] Synchronization overhead in single-threaded applications can be measured by taking the difference between the execution times of a program with and without synchronization operations. This experiment cannot be performed on multithreaded programs because they do not run correctly without synchronization.

| Benchmark | Description | Bytecode size (000s bytes) | | Classes (number) | | Methods (number) | |
|---|---|---|---|---|---|---|---|
| | | Application | Library | Application | Library | Application | Library |
| Single-threaded programs | | | | | | | |
| cassowary | Constraint solver (UW) | 79 | 989 | 29 | 430 | 459 | 6748 |
| javac | Source-to-bytecode compiler for Java (Sun) | 624 | 1015 | 173 | 444 | 2212 | 6914 |
| javacup | Parser-generator for Java | 130 | 989 | 34 | 430 | 509 | 6748 |
| javadoc | Documentation generator for Java | 675 | 1013 | 177 | 445 | 2360 | 6936 |
| jgl | Java Generic Library array benchmarks | 874 | 989 | 262 | 430 | 5216 | 6748 |
| jlex | Lexical analyzer for Java | 91 | 989 | 20 | 430 | 213 | 6748 |
| pizza | Source-to-bytecode compiler for Java | 819 | 1004 | 239 | 438 | 3203 | 6851 |
| Multithreaded programs | | | | | | | |
| array | Parallel matrix multiplication | 46 | 989 | 29 | 430 | 178 | 6748 |
| instantdb | Database with a TPC-A-like workload | 307 | 2447 | 67 | 956 | 1268 | 15891 |
| jlogo | Multithreaded Logo interpreter | 202 | 989 | 58 | 430 | 760 | 6748 |
| jws | Web server (Sun) | 564 | 3142 | 510 | 903 | 2743 | 19363 |
| plasma | plasma simulation | 8 | 2016 | 1 | 1161 | 19 | 18193 |
| proxy | Network proxy for the HTTP protocol | 7 | 989 | 3 | 430 | 21 | 6748 |
| raytrace | Ray tracer with geometric objects | 29 | 1313 | 18 | 536 | 190 | 8688 |
| slice | visualization tool | 23 | 2016 | 13 | 1161 | 71 | 18255 |

**Table 1: Characteristics of our Benchmark Suite**

### 2.3 Unnecessary Synchronization Frequency by Type

In order to determine the potential benefits of optimizing each type of unnecessary synchronization, we studied the synchronization characteristics of a diverse set of Java programs. Table 1 shows the broad scope of our benchmark suite, which includes seven single-threaded and eight multithreaded programs of varying size. Our applications are real programs composed of 20 to 510 classes, in domains ranging from compiler tools to network servers to database engines. We consciously chose multithreaded programs, because they cannot be trivially optimized by removing all synchronization. The programs in our suite include some of the largest Java programs publicly available, allowing us to demonstrate the scalability of our techniques.

To assess the relative importance of each type of unnecessary synchronization, we used execution traces to measure the dynamic frequency of each type for each of our 15 benchmark programs. A synchronization operation is dynamically thread-local if the corresponding lock is only used by one thread during program execution. A synchronization operation is dynamically enclosed if all operations on its lock occur while some other lock is locked. Finally, a synchronization operation is dynamically reentrant if its lock is already locked when the operation occurs. A given synchronization operation can fall into more than one category, so the total percentage of unnecessary synchronization is typically less than the sum of the

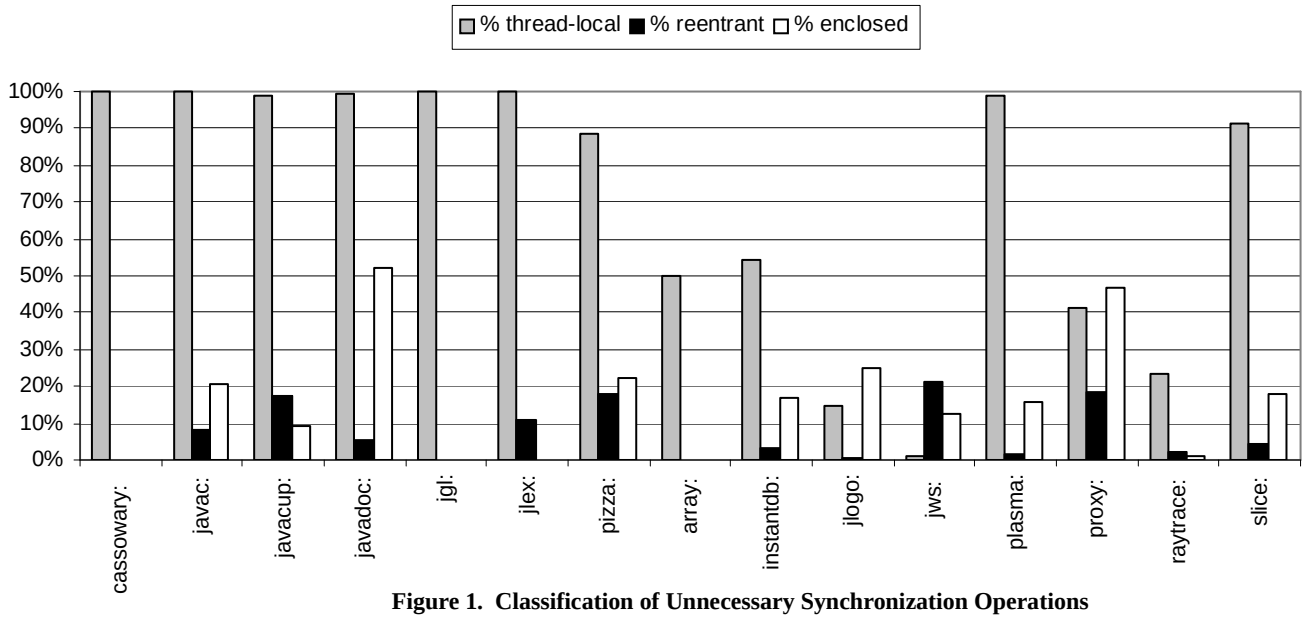


**Figure 1. Classification of Unnecessary Synchronization Operations**

contributions from each category. This also implies that analyses that focus on different kinds of unnecessary synchronization may optimize some of the same synchronization operations.

Figure 1 shows that all three types of unnecessary synchronization occur frequently in some benchmarks. These figures represent an optimistic upper bound on how well any static analysis could eliminate a particular type of unnecessary synchronization. The most common type is thread-local synchronization: all synchronization (except for a finalizer thread) is thread-local for the single-threaded benchmarks, and a significant fraction of synchronization is thread-local even for the multithreaded programs. Enclosed synchronization makes an important contribution for `javadoc`, `proxy`, and a number of the other benchmarks. Finally, reentrant synchronization makes a small contribution to many different benchmarks.

## 3. Related Work

The rapid deployment and acceptance of Java, with its multi-threaded programming model and support for lock synchronization, has recently fueled research on eliminating unnecessary synchronization operations. While the proposed analyses and optimization techniques have been quite diverse, they have all targeted a single source of unnecessary synchronization, that of thread-local objects. Thread-local locks are accessed by at most one thread and once identified can be eliminated via specialization or by explicit checks.

Blanchet [B99] identifies thread-local objects through escape analysis by encoding reference and subtyping relationships with integer type heights. A flow-insensitive analysis is used both to allocate thread-local objects on the stack and to eliminate

synchronization from stack-allocated objects. Stack-allocated objects are marked, and each synchronization operation checks whether an object is on the stack before locking it. In addition, this optimization modifies method invocations to call an unsynchronized version of a method when the receiver object does not escape.

Bogda and Hölzle [BH99] have also defined a flow-insensitive escape analysis to eliminate synchronization from thread-local objects. The analysis is limited to thread-local objects that are only reachable by paths of one or two references from the stack. It removes synchronization by specializing classes to create a subclass with unsynchronized methods, and modifying allocation sites of thread-local objects to use the unsynchronized versions. The analysis may also clone several methods leading to an allocation site, enabling it to distinguish thread-local and multithreaded objects created at the same program point.

Choi et al. [CGS+99] performs a variant of interprocedural points-to analysis that is designed to work well when classifying objects as globally escaping, escaping via an argument, and not escaping. The analysis groups objects by their allocation site and marks thread-local objects at allocation time with a bit in the object header. When synchronizing, the compiler eliminates the atomic compare-and-swap operation for objects with this bit in the header, preserving Java semantics by flushing the local processor cache. The analysis also allocates objects on the stack.

Whaley and Rinard [WR99] define an interprocedural, flow-sensitive points-to analysis to eliminate unnecessary synchronization and allocate objects on the stack. The analysis computes which objects escape from each method, as well as relationships between objects. It can analyze partial programs conservatively, improving results as more of the program becomes available. When the analysis finds that an object is captured by a method, it specializes all methods that synchronize on that object in order to remove the synchronization. It also generates specialized versions of all methods in the call chains that lead to the optimizable synchronized method invocations.

Other researchers have attacked the cost of synchronization in different ways. A large body of work [BKMS98][KP98][ADG+99][OK99] has focused on making necessary synchronization more efficient, which complements our techniques to remove unnecessary synchronization. It is also possible to optimize unnecessary synchronization that arises from acquiring and releasing a lock multiple times in succession [DR98][PZC95]. These optimizations affect the concurrency of the benchmarks: coalescing multiple lock operations into one may reduce parallelism, and implementations must take care not to introduce deadlock. Finally, some systems perform synchronization analyses to help programmers model concurrent systems [C98] or to help find synchronization errors [DLNS98].

Our research differs from this previous work in several important respects:

- First, all previous studies of unnecessary synchronization have concentrated solely on eliminating thread-local locks. As shown in section 2.3, thread-local locks are only one of several sources of unneeded synchronization. In this paper, we address three types, presenting new algorithms for eliminating two of them (thread-local and enclosed locks) and empirically evaluating all three.
- Second, previous analyses for identifying thread-local objects have relied on escape analyses that ignore the way programs use concurrency. Our thread-local algorithm explicitly models the interactions between different threads, and we quantify the resulting improvement over earlier thread-oblivious algorithms; our algorithms do better than previous work on all benchmarks we have in common.
- Finally, previous evaluations of optimization schemes have relied predominantly on single-threaded benchmarks. While improving the performance of single-threaded programs is important, a trivial optimization to simply disable all synchronization is most effective. Thus an important benefit of a synchronization elimination algorithm comes from distinguishing unnecessary synchronization operations from necessary ones in multithreaded programs. In this paper, we evaluate our analyses on *both* single-threaded and multithreaded applications, and quantify the difference in our analyses' performance on the two application types.

Concurrent work by Eric Ruf [R00] combines a thread behavior analysis similar to ours with a specialized alias analysis based on method summaries. His specialized alias analysis is more scalable than the general-purpose analyses in our system, resulting in much smaller analysis times. His results for thread-local synchronization are similar to ours for the small programs we have in common, as well as the larger benchmarks `plasma` and `javac`. Ruf's alias analysis enabled him to remove significant amounts of synchronization from `slice`, while our precise alias analysis did not scale to this benchmark, resulting in poor performance. Ruf's work does not consider the other forms of unnecessary synchronization, enclosed and reentrant locks.

## 4. Analyses

We define a simplified analysis language and describe three analyses necessary to optimize the synchronization opportunities discussed above: thread-local analysis, reentrant and enclosed lock analysis, and unshared field analysis. Thread-local analysis identifies which objects are only synchronized by one thread. Lock analysis computes a description of the monitors held at each synchronization point so that reentrant locks and enclosed locks can be eliminated. Finally, unshared

```
id, f, field ∈ ID
label, base, o ∈ LABEL
s ∈ S
p, program ∈ P

P ::=
   P ; P
   letrec id_F := λ(id_1..id_n) { S }

S ::=
   id := new^label
   id_1 := id_2.f
   id_1.f := id_2
   S ; S
   if id_1 then S_1 else S_2
   fork id_F^label()
   synchronized^label(id) { S }
   id_0 := id_F(id_1..id_n)^label
```

**Figure 2. Simplified analysis language incorporating the key synchronization features of Java**

field analysis identifies unshared fields so that lock analysis can safely identify enclosed locks. 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 statement-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. The missing features of Java can be mapped to our core language. For example, loops can be converted into recursion, method dispatch can be implemented with if statements, variable assignment can be done with variable renaming, exceptions can be emulated using if statements, etc. These features do not present any difficult problems for our analysis, but would make the presentation more complex.

Figure 2 presents our analysis language. It is a simple, first-order language, incorporating object creation, field access and assignment, synchronization expressions, threads, functions, and simple control flow. Each object creation point is labeled with a label for 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 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.

Functions are modeled with a `letrec` construct and uniquely labelled function calls. Return values are implemented by assigning to the special variable `returnval`. Threads are modeled with a labelled fork statement that starts the indicated function in a new thread. Java's synchronization construct is modeled by a **`synchronized`** statement, which locks the object referred to by `id` and then evaluates `S` before releasing the lock. Each **`synchronized`** statement in the program text is also associated with a unique label.

### 4.2 Analysis Axioms

Our analyses are parameterized by other alias and call-graph construction analyses, a feature of our approach that allows a trade-off between analysis time and the precision of our analysis results. We assume the following axioms are defined from earlier analysis passes:

- *aliases*(`id`$_1$, `id`$_2$) – identifiers `id`$_1$ and `id`$_2$ may point to the same object
- *aliases*(`f`$_1$, `f`$_2$) – fields `f`$_1$ and `f`$_2$ may point to the same object
- *ref*(`base`, `f`, `o`) – objects created with the new statement labeled `o` may be stored in the field f of objects with label `base`
- *creator*(`o`) – the procedure that created the objects identified by label o
- *ref*(`id`, `o`) – identifier `id` may refer to objects labeled o
- *synch_aliases*(`label`) – the set of labels of synchronization points that may synchronize the same object as the synchronization point identified by `label`
- *synch_keys*(`label`) – the set of objects that may be synchronized at synchronization point `label`
- *immutable*(`f`) – field `f` is immutable (i.e., write-once). This may be deduced from **`final`** annotations and constructor code.
- *called*(`p`, `label`$_q$) – function `p` may be called from call site `label` in function `q`. This relation includes forked functions as well as ordinary function calls.

### 4.3 Thread-local Analysis

Thread-local analysis examines the behavior of threads in a program to identify objects that are not accessed by more than one thread. In Java, there are just two ways to share an object between threads. First, an object can be written to a static field by one thread and then read from that field by another. Second, a thread can write an object to a (non-static) field of an object that is or will be shared by another thread, and the second thread can then read the object from that field. By looking at how these two mechanisms are used in a particular program, the analysis discovers the set of *multithreaded objects*, *i.e.* objects that are shared between threads. Objects *not* in this set are thread-local.

We present our analysis in figure 3, using inference rules. Our analysis starts with an axiom representing the execution of the `main` function in the initial "`main`" thread:

- *eval*(`main()`, `main`)

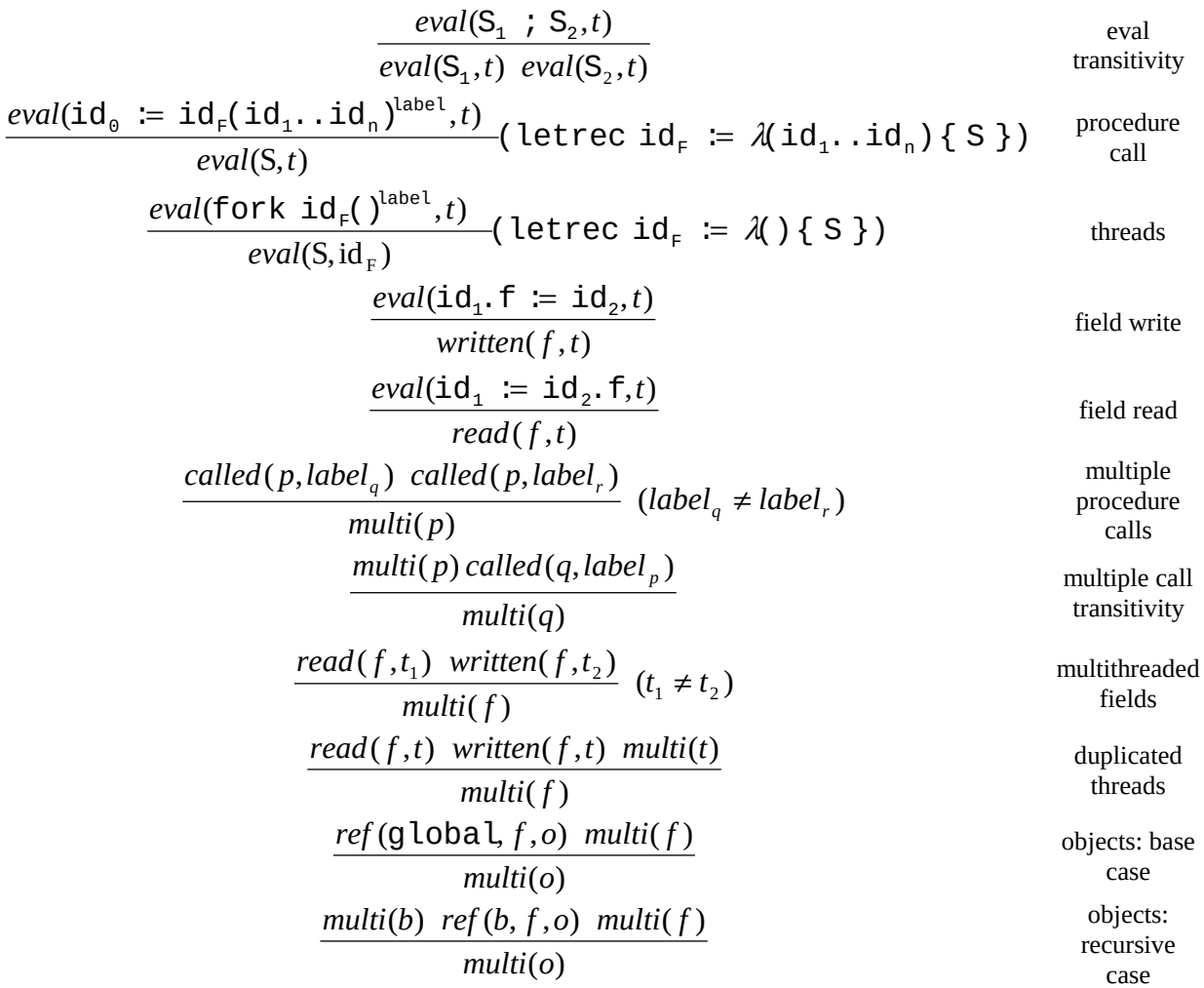


**Figure 3. Inference rules describing Thread-local Analysis**

The result of inference will be the least set of judgments closed under application of these inference rules to the axioms above. Note that a thread is represented by the forked procedure, and so the set of threads is a subset of the set of procedures. The facts that can be inferred from our inference rules include:

- *eval*(s, t) – statement s may be executed inside thread t
- *read*(f, t) – field f may be read by thread t
- *written*(f, t) – field f may be written by thread t
- *multi*(p) – procedure p may be called more than once
- *multi*(t) – thread t may be started more than once
- *multi*(f) – field f may be read by one thread and written by another thread

- *multi*(o) – object o may be accessed by more than one thread. This is the main result of our thread-local analysis.

At a high level, the thread-local algorithm takes advantage of the two simple sharing mechanisms by analyzing which threads read and write to which fields. We call a field is that read by one thread and written by another a *multithreaded field.* Our algorithm computes the set of multithreaded fields by determining the set of threads that may execute each field read and write in the program. For each static thread instance (represented by a forked procedure), we also need to determine whether it is started more than once, because if a thread with multiple dynamic instances both reads and writes to a field, that field will be multithreaded. Thus we must scan the program to find out which thread forking statements may execute more than once.

Thread-local analysis runs in worst case $O(o^2*f + n^2*t)$ time and $O(n*t)$ space, where o is the number of object sets, f is the number of fields per object, t is the number of forked procedures, and n is the size of the program. This property can be deduced from our analysis rules using McAllester's technique [M99]. In general, this is $O(n^3)$ time and $O(n^2)$ space, where n is the program size. However, in practice the analysis scales linearly with the number of application classes analyzed, probably because the number of static thread instances, the number of fields per object, and the virtual method dispatch fan-out tend to be small in typical Java programs. Our thread-local analysis runs quickly, typically completing in far less time than the alias analyses we run beforehand.

Our new thread-local analysis differs from our previous work [ACSE99] in that it considers thread interactions to intelligently decide which fields allow objects to be shared between different threads. Our previous analysis, like other previous work in the field, was overly conservative in that it assumed that all fields are multithreaded.

### 4.4 Lock Analysis

An enclosed lock (say, $L_2$) is a lock that is only acquired after another (enclosing) lock $L_1$ has already been locked. If all threads follow this protocol, synchronization operations on $L_2$ are redundant and can be eliminated. Enclosed locks occur often in practice, particularly in layered systems, generic libraries or reusable object components, where each software module usually performs synchronization independently of other modules. Established concurrent programming practice requires that programs acquire locks in the same global order throughout the computation in order to avoid deadlock. Consequently, most well-behaved programs exhibit a self-imposed lock hierarchy. The task of this analysis, then, is to discover this hierarchy by simulating all potential executions of the threads, identify the redundant lock operations and optimize them out of the program.

We rely on a flow-sensitive, interprocedural analysis in order to eliminate locks that are protected from concurrent access by other locks. Our analysis works by calculating the set of enclosing locks for each lock in the program; reentrant locks

represent a special case where the enclosing lock is the same as the enclosed lock. This set of enclosing locks is computed by traversing the call graph, starting from each thread's starting point. Whenever a lock acquire or release operation is encountered, the locked object is added to or deleted from the set of locks currently held at that program point. In order to permit specialization based on the creation points of program objects, our algorithm is context sensitive and thus will analyze a method once for every calling context (i.e., set of possible receiver and argument objects).

When removing synchronization due to enclosing locks, it is crucial that there be a unique enclosing lock; otherwise, the enclosing lock does not protect the enclosed lock from concurrent access by multiple threads. Because one static lock may represent multiple dynamic locks at runtime, we must ensure that a unique dynamic lock encloses each lock eliminated by the analysis. We can prove this in multiple ways. First, in the reentrant case the locks are identical, as when the same variable is locked in a nested way twice, without an assignment to that variable in between. Second, a lock may be enclosed by an object who's creation point is only executed once (as calculated by the thread-local analysis); thus a single static lock represents a single dynamic lock. Third, the enclosed lock may hold the enclosing lock in one of its fields; this field must be immutable, to ensure that the following the field link uniquely specifies the enclosing lock. Fourth, the enclosing lock may hold the enclosed lock in one of its fields. In this case, immutability is not important, because a single enclosing lock may protect multiple enclosed locks; however, a corresponding property is required. The enclosing lock's field must be unshared, indicating that the object held in the field is never held by any other object in the same field; thus the enclosing object is unique with respect to the enclosed object. Section 4.5 presents an analysis that finds unshared fields. Finally, the last two cases can be generalized to a path along field links from one object to another, as long as each field in the path is immutable or unshared, depending on the direction on which the path traverses that link.

Our lock analysis represents a reentrant lock as the synchronization expression itself (SYNCH), and enclosing locks are represented as unique objects (denoted by their creation-point label) or as paths from the synchronization expression SYNCH through one or more field links to the destination object. We use a Link Graph (LG) to capture relationships between different identifiers and locks in a program. The LG is a directed graph with nodes labeled with unique identifiers or placeholders, and edges labeled with a field identifier. We notate functional operations on the graph as follows:

- Adding an edge: newgraph = *add*(graph, $\texttt{id}_1 \rightarrow_{\texttt{f}} \texttt{id}_2$)
- Replacing nodes: newgraph = graph[$\texttt{id}_1 \rightarrow \texttt{id}_2$]
- Treeshake: newgraph = *treeshake*(graph, rootset)

- Union: newgraph = $graph_1 \cup graph_2$
- Intersection: newgraph = $graph_1 \cap graph_2$

The *treeshake* operation removes all nodes and edges that are not along any directed path connecting a pair of nodes in the root node set. The union operation merges two graphs by merging corresponding nodes, and copying non-corresponding edges and placeholder nodes from both graphs. The intersection operation is the same as union, except that only edges and placeholder nodes that are common to the two graphs are maintained. In these operations, two nodes *correspond* if they have the same label or are pointed two by identical links from corresponding nodes.

Intuitively, an edge in the link graph means that at some program point there was an immutable field connecting the edge source to the edge destination, or there was an unshared field connecting the edge destination to the edge source. Note that edges representing immutable fields go in the opposite direction as edges representing unshared fields; this captures the notion that the two are really inverses for the purposes of our enclosing lock analysis. The link graph does not strictly represent an alias graph, as we do not kill any edges on updates. This is acceptable because the link graph only contains edges where updates don't matter (unshared fields) or can't occur (immutable fields).

Figure 4 presents our lock analysis as a semantic analysis over the program text. The analysis function L accepts as curried parameters a bit of program text, the set of locks held at the current program point, and a link graph. The analysis function manipulates the inputs according to the form of the program text and returns an updated link graph. The function also updates a global data structure that tracks the set of enclosing locks at each synchronization point. Analysis is triggered through the *get_locks* function, which runs the analysis on main assuming an empty lock set and link graph (as well as optimistic assumptions about enclosing locks at each synchronization point). *Get_locks* then looks up the set of locks at the relevant synchronization point in the data structure produced as an analysis side effect. During the analysis, a set of locks is represented by a LOCKSET structure, which is a set of nodes in a link graph. In the data structure lockmap, which maps synchronization points to the set of locks held at each point, locks are represented as a PATH in a link graph, where the source node has been either replaced with the node SYNCH (representing the current synchronization expression) or with the label of a unique object.

The rule for `new` does not affect any data structures. Field reads and writes are equivalent for our analysis: if the field is unshared or immutable, then a link is established between the identifiers in the appropriate direction. Analyzing a sequence of statements simply analyzes the first and uses the resulting link graph to analyze the second. After an if statement, it is only safe

$L$ : ⟦syntax⟧ → LOCKSET → LG → LG

// all global tables are initializes to optimistic values (- )
**global** context_table : CONTOUR →$_{fin}$ $\text{LOCKSET}_{in} \times \text{LG}_{in} \times \text{LG}_{out}$
**global** lockmap : LABEL × CONTOUR →$_{fin}$ $2^{\text{PATH}}$
**global** enclosingmap : LABEL →$_{fin}$ $2^{\text{PATH}}$

*get_locks*(label) : LOCKSET =
**let** ignore = $L$⟦main()⟧Ø Ø **in** lockmap(label)

$L$⟦$\text{id}$ := **new**$^{\text{label}}$⟧lockset lg = lg
$L$⟦$\text{id}_2$ := $\text{id}_1$.f⟧lockset lg =
**let** lg' = **if** *is_immutable*(f) **then** *add*(lg, $\text{id}_1 \rightarrow_f \text{id}_2$) **else** lg **in**
**if** *is_unshared*(f) **then** *add*(lg', $\text{id}_2 \rightarrow_f \text{id}_1$) **else** lg'
$L$⟦$\text{id}_1$.f := $\text{id}_2$⟧lockset lg =
**let** lg' = **if** *is_immutable*(f) **then** *add*(lg, $\text{id}_1 \rightarrow_f \text{id}_2$) **else** lg **in**
**if** *is_unshared*(f) **then** *add*(lg', $\text{id}_2 \rightarrow_f \text{id}_1$) **else** lg'
$L$⟦$S_1$ ; $S_2$⟧lockset lg =
**let** lg' = $L$⟦$S_1$⟧lockset lg **in**
$L$⟦$S_2$⟧lockset lg'
$L$⟦**if** $\text{id}_1$ **then** $S_1$ **else** $S_2$⟧lockset lockmap lg =
**let** lg' = $L$⟦$S_1$⟧lockset lg **in let** lg'' = $L$⟦$S_2$⟧lockset lg **in** lg' ∩ lg''
$L$⟦**fork** $\text{id}_F$()⟧lockset lg =
**let** ⟦**letrec** $\text{id}_F$ := λ() { S }⟧= *lookup*($\text{id}_F$) **in**
**let** ignore = $L$⟦S⟧Ø Ø **in**
lg
$L$⟦**synchronized**$^{\text{label}}$(id) { S }⟧lockset lg =
**let** unique_sources = { id } ∪ { s | s ∈ lockset ∧ **not** *multi*(*creator*(s)) } **in**
**let** locks = { path[id → **SYNCH**] | path ∈ lg ∧ *source*(path) ∈ unique_sources ∧ *destination*(path) ∈ lockset } **in**
lockmap := lockmap[(label, *current_contour*())→ locks ∩ lockmap[(label, *current_contour*())]];
enclosingmap := enclosingmap[ o → locks ∩ enclosingmap[o] | ref(id, o) ]
**let** object_refs = { o | ref(id, o) } **in**
**let** unique_locks = **if** object_refs = {o} ∧ *multi*(*creator*(o)) **then** {o} **else** Ø **in**
$L$⟦S⟧ (lockset ∪ { id } ∪ unique_locks) lg
$L$⟦$\text{id}_0$ := $\text{id}_F$($\text{id}_1$..$\text{id}_n$)$^{\text{label}}$⟧lockset lg =
**let** ⟦**letrec** $\text{id}_F$ := λ($\text{formal}_1$..$\text{formal}_n$) { S }⟧= *lookup*($\text{id}_F$) **in**
**let** lg' = *treeshake*(lg, { $\text{id}_1$..$\text{id}_n$ } ∪ lockset ) **in**
**let** mapping = { oldnode → *new_placeholder*() | oldnode ∈ *nodes*(lg) } **in**
**let** mapping' = mapping[$\text{id}_i$ → $\text{formal}_i$ | i ∈ 1..n] **in**
**let** lg'' = *context_strategy*($\text{id}_F$, lockset[node → mapping'[node]], lg'[node → mapping'[node]]) **in**
lg ∪ lg'' [node → original | (original → node) ∈ mapping' ] [returnval → $\text{id}_0$]
*context_strategy*($\text{id}_F$, lockset, lg) =
**let** ⟦**letrec** $\text{id}_F$ := λ($\text{formal}_1$..$\text{formal}_n$) { S }⟧= *lookup*($\text{id}_F$) **in**
**let** contour = *get_contour*(*meta information*) **in**
**let** (prev_lockset, prev_lg, prev_result) = context_table[contour] **in**
**if** (prev_lockset ⊆ lockset ∧ prev_lg ⊆ lg) ∨ *is_recursive_call*() **then** prev_lg
**else let** lockset' = lockset ∩ prev_lockset **in**
**let** lg' = lg ∩ prev_lg **in**
**let** lg'' = $L$⟦S⟧lockset' lg' **in**
**let** lg''' = *treeshake*(lg'', { $\text{formal}_1$..$\text{formal}_n$ } ∪ { returnval } ∪ lockset) **in**
context_table := context_table[contour → (lockset', lg', lg''') ];
lg'''

**Figure 4. Semantic analysis functions for Lock Analysis**

to assume relationships established along both paths, so the resulting link graph is the intersection of the link graphs from the then and else clauses. A fork statement begins analysis in the new thread with an empty lock set and link graph.

At synchronization statements, the analysis records all enclosing locks at that statement, and adds the locked object to the set of locks. For an enclosing lock to be uniquely specified with respect to the locked expression, it must begin with a unique expression, which may be a singleton object (created only once during program execution) or may be the locked object itself. From this unique object, we can find a path through the link graph, where each link uniquely specifies its destination with respect for the source due to the properties of the graph. If the final object in the path is in the lock set, the path describes a unique enclosing lock, and therefore it is added to the lockmap for that synchronization expression. The locks determined by this analysis are intersected with all other analyses of the same contour, and the result is saved for this synchronization point and contour. We also save the enclosing lock set for each object the identifier may refer to, intersecting it with the previous set of enclosing locks for that object. Then the identifier itself is added to the lock set for evaluation of the synchronization block, and if this identifier points to a single, unique object, the representation for that object is added to the lock set as well.

At function calls, the analysis finds the formals of the called function and maps identifiers to formals. Only the part of the link graph that links formal identifiers and locked objects is relevant to the callee, so all other parts of the graph are removed. Nodes representing identifiers that are no longer in scope in the new function are replaced with placeholder nodes at the same location in the graph; such nodes may represent locked objects, or may be along a unique path to locked objects. The callee is analyzed using one of several possible context strategies, and a reverse mapping applies the resulting link graph to the caller.

In our implementation, the context strategy analyzes each function multiple times according to the calling context, which may be represented by the classes of the arguments, the calling function, or other meta-information. In our implementation, we used the sets of argument classes from the SCS call-graph construction algorithm as our contours. A contour table caches (input, output) analysis pairs for each contour, to avoid excessive contour re-analysis and handle recursion appropriately. If the input information from the contour table is a conservative approximation of the current input information, the old output information is returned. For recursive calls to the same contour, an optimistic initial result is returned, and the framework will automatically re-analyze the callee when that optimistic result is later modified, preserving analysis soundness. Finally, the text of the new procedure is analyzed, the result is combined with previous results, cached in the context table, and returned to the analysis of the callee.

fset ∈ FSET = $2^{ID}$
idstate ∈ IDSTATE = ID $\rightarrow_{fin}$ FSET

*U* : ⟦syntax⟧ → IDSTATE → IDSTATE

// all global tables are initializes to optimistic values (- )
**global** shared : FSET;
**global** context_table : CONTOUR $\rightarrow_{fin}$ $IDSTATE_{in} \times IDSTATE_{out}$;

*is_unshared*(field)= **let** ignore = *U* ⟦main()⟧Ø **in** (field ∉ shared)

*U*⟦$id_1$ := **new**$^{label}$⟧idstate = idstate[$id_1$ → Ø]
*U*⟦$id_1$ := $id_2$.f⟧idstate = idstate[$id_1$ → { f' | *aliases*(f, f') }]
*U*⟦$id_1$.f := $id_2$⟧idstate = **let** fset = { f } ∪ idstate[$id_2$] **in**
  **if** f ∈ idstate[$id_2$] **then** shared := shared ∪ { f };
  idstate[id → {f} ∪ idstate[id] | *aliases*(id, $id_2$)]
*U*⟦$S_1$ ; $S_2$⟧idstate = **let** idstate' = *U*⟦$S_1$⟧idstate **in** *U*⟦$S_2$⟧idstate'
*U*⟦**if** $id_1$ **then** $S_1$ **else** $S_2$⟧idstate =
  **let** idstate' = *U*⟦$S_1$⟧idstate **in let** idstate" = *U*⟦$S_2$⟧idstate **in** idstate' ∪ idstate"
*U*⟦**fork** $id_F$()⟧idstate =
  **let** ⟦**letrec** $id_F$ := λ() { S }⟧ = *lookup*($id_F$) **in**
  **let** ignore = *L*⟦S⟧Ø;
  idstate
*U*⟦**synchronized**$^{label}$(id) { S }⟧idstate = *U*⟦S⟧idstate
*U*⟦$id_0$ := $id_F$($id_1$..$id_n$)$^{label}$⟧idstate =
  **let** ⟦**letrec** $id_F$ := λ($formal_1$..$formal_n$) { S }⟧ = *lookup*($id_F$) **in**
  **let** idstate' = {$formal_i$ → idstate[$id_i$] | i ∈ 1..n} **in**
  **let** idstate" = *context_strategy*($id_F$, idstate') **in**
  **let** newidstate = idstate[$id_0$ → idstate"[returnval]] **in**
    newidstate[id → idstate"[$formal_i$] ∪ idstate[id] | i ∈ 1..n ∧ *aliases*(id, $id_i$)]
*context_strategy*($id_F$, idstate) =
  **let** ⟦**letrec** $id_F$ := λ($formal_1$..$formal_n$) { S }⟧ = *lookup*($id_F$) **in**
  **let** contour = *get_contour*(*meta information*) **in**
  **let** (prev_idstate, prev_result) = context_table[contour] **in**
    **if** prev_idstate ⊇ idstate ∨ *is_recursive_call*() **then** prev_result
    **else let** idstate' = idstate ∪ prev_idstate **in**
      **let** idstate" = *U*⟦S⟧idstate' **in**
        context_table := context_table[contour → (idstate', idstate")];
        idstate"

**Figure 5. Semantic analysis functions for Unshared Field Analysis**

### 4.5 Unshared Field Analysis

Unshared field analysis detects fields that uniquely enclose the object they point to. They provide a natural counterpart to immutable fields (including **final** fields) which uniquely point to a particular object (which can then be used as an enclosing lock). Figure 5 shows our flow- and context-sensitive analysis to detect unshared fields. The basic principle of the analysis is to conservatively track the set of fields that each identifier could possibly alias with. Thus the analysis passes around and updates a mapping from the identifiers in scope to the set of possibly aliased fields. New objects do not alias any fields, but an

assignment of a field dereference to an identifier means that identifier may be aliased to the dereferenced field, or any field the dereferenced field may alias. When a field is assigned an identifier's value, we check whether the identifier could already include that field; if so, we have identified a case where an object may be shared between two instances of the same field. That field becomes *shared* and is added to the shared field set. Meanwhile, the identifier (and any other identifier that may point to the same object) may be aliased to the assigned field.

Note that if an object is assigned to two different fields, neither field becomes shared; fields only become shared when the same object is possibly assigned to two instances of the same field. This does not lead to incorrect results in lock analysis, because field links are annotated with the field name, allowing the two different enclosing objects to be distinguished.

A sequence of statements is evaluated in turn, and control flow merges use union to conservatively combine the identifier state along each path. Synchronization statements do not affect identifier state. At function calls, a fairly straightforward actual to formal mapping is applied in a similar style to lock analysis. The resulting callee's identifier state must be combined with the caller's identifier state taking identifier aliases into consideration, because the arguments and results may be assigned to fields within the callee. Finally, our context strategy for this analysis is similar to that for lock analysis, except that the calling context is simply the input identifier state.

### 4.6 Optimizations

We apply three optimizations for the three cases of unnecessary synchronization. We test each synchronization statement for thread-local synchronization. If there is no conclusion *multi*(o) from thread-local analysis for any o possibly synchronized by a synchronization statement s, then s can be removed from the program. Otherwise, if there is any o synchronized at s for which there is no conclusion *multi*(o), and the synchronized object is the receiver of the method (the common case, including all synchronized methods), then a new version of the method is cloned for instances of o without synchronization. This specialization technique may require cloning o's class, and changing the new statements that create instances of o to refer to the new class. Our implementation choice of sets of receiver classes as contours allows us to naturally use our analysis information when specializing methods.

We also test each synchronization statement for reentrant synchronization. For an synchronization expression s, if the lock set includes **SYNCH**, then s can be removed from the program. If **SYNCH** is in the lock set for some receivers but not for others, specialization can be applied using the technique above.

Finally, we can remove a synchronization statement s if, for each object o synchronized at s, the set of enclosing locks given by enclosingmap[o] is not empty. If only some of the objects synchronized at s are enclosed, and synchronization is on the receiver object, the method can be specialized in a straightforward way to eliminate synchronization.

### 4.7 Implementation

Our analyses are implemented in the Vortex research compiler [DDG+96], which performs whole program analysis. Our analyses assume that a call-graph construction and alias analysis pass has been run. In our implementation, we use the Simple Class Sets (SCS) context-sensitive call graph construction algorithm [GDDC97] for the smaller benchmarks, and the context-insensitive 0-CFA algorithm [S88] for the benchmarks larger than javac. We augmented the algorithms to collect alias information based on the creation points of objects. While these algorithms build a precise call graph, they compute very general alias information and as a result the SCS algorithm did not scale to our largest benchmarks. An alias analysis specialized for synchronization elimination [R00] would allow large improvements in scalability and analysis performance, at the potential cost of not being reusable for other compiler analysis tasks. The analyses above are fully implemented in our system, except that we use an approximation of the link graph and do not consider immutable fields; these pieces do not make a significant contribution to unnecessary synchronization in our benchmarks.

In order to show the effectiveness of our analyses on a typical runtime platform, we sent our analysis results to a binary rewriter [SGGB99] that performs optimizations on Java class files. These optimized application classes can then be run on any Java virtual machine.

A production compiler that targets multiprocessors would still have to flush the local processor cache at eliminated synchronization points in order to conform to Java's memory model [P99]. Due to our binary rewriting strategy, we could not implement this technique in our system.

## 5. Results

In this section, we evaluate the performance of our analyses. Section 5.2 shows that they can improve the performance of a workload in which programmers had eliminated synchronization manually; section 5.1 demonstrates their potential for enabling a simpler and more general synchronization model; and section 5.3 describes their compile-time cost.

### 5.1 Dynamic Evaluation of the Synchronization Analyses

In this section we evaluate the impact of our analyses on the dynamic behavior of the benchmarks. Table 3 shows the dynamic percentage of synchronization operations eliminated at runtime by our analyses. The first column represents the

| Benchmark | All | Thread-Local | | Reentrant | | Enclosed | | Total Ops | Ops/sec |
|---|---|---|---|---|---|---|---|---|---|
| | Actual | Actual | Potential | Actual | Potential | Actual | Potential | | |
| cassowary | 99.98% | 99.98% | 99.99% | 0.00% | 0.01% | 0.00% | 0.06% | 4440928 | 25715 |
| javac | 94.55% | 94.55% | 99.79% | 0.02% | 8.14% | 0.00% | 20.54% | 1378442 | 49373 |
| javacup | 78.12% | 78.12% | 99.08% | 2.60% | 17.57% | 0.00% | 9.02% | 19174 | 4117 |
| javadoc | 82.76% | 82.76% | 99.66% | 0.05% | 5.60% | 0.00% | 52.42% | 909490 | 22689 |
| jgl | 99.99% | 99.99% | 100.00% | 0.00% | 0.00% | 0.00% | 0.00% | 5529820 | 162766 |
| jlex | 99.95% | 99.95% | 99.99% | 4.37% | 11.14% | 0.00% | 0.07% | 1839166 | 276442 |
| pizza | 64.26% | 64.26% | 88.36% | 0.61% | 18.10% | 0.00% | 22.29% | 20125 | 6492 |
| array | 44.44% | 44.44% | 50.12% | 0.02% | 0.12% | 0.00% | 0.13% | 90693 | 372 |
| instantdb | 0.01% | 0.00% | 54.31% | 0.01% | 3.43% | 0.00% | 16.78% | 302640 | 27143 |
| jlogo | 12.03% | 0.21% | 14.85% | 0.08% | 0.37% | 11.81% | 25.26% | 164501 | 1871 |
| jws | 0.01% | 0.00% | 0.83% | 0.01% | 21.42% | 0.00% | 12.48% | 1062766 | N/A |
| plasma | 89.30% | 89.25% | 98.87% | 0.05% | 1.36% | 0.00% | 15.91% | 34723 | 62 |
| proxy | 43.29% | 39.45% | 41.43% | 3.84% | 18.34% | 0.00% | 46.77% | 364624 | N/A |
| raytrace | 72.78% | 72.69% | 96.00% | 0.19% | 2.04% | 0.00% | 1.22% | 34351 | N/A |
| slice | 0.08% | 0.00% | 91.22% | 0.08% | 4.23% | 0.00% | 17.86% | 39533 | 255.006 |

**Table 2: Dynamic Number of Synchronization Operations Eliminated**

percentage of runtime synchronization operations removed by all of our analyses combined. The next three pairs of columns break this down into thread-local, reentrant, and enclosed locks. The first column in each pair shows the percentage of locks in each category that is optimized by its appropriate analysis, while the second column in the pair is the total amount of dynamic synchronization in the category, as measured by the dynamic traces (it thus serves as an upper bound on the analysis results). (Recall that, since many synchronization operations fall into several categories, the totals of each pair do not sum to 100%; in particular, many enclosed and reentrant locks are also thread-local.) Finally, the last two columns show the total number of lock operations and the frequency in operations per second.

In general, thread-local analysis did well for most of the benchmarks, eliminating a majority (64-99+%) of synchronization operations in our singlethreaded benchmarks and a more widely varying percentage (0-89%) of synchronization in our multithreaded applications. Among the single-threaded programs, it optimized `jlex`, `cassowary`, and `jgl` particularly well, eliminating over 99.9% of synchronization in these programs. We also eliminated most thread-local synchronization in the other singlethreaded programs, but did not realize the full potential of our analyses. In the multithreaded programs, where the challenge is greater, the thread-local analysis usually did well, getting most of its potential for `array`, `proxy`, `plasma` and raytrace. Our dynamic traces show very little thread-local synchronization in `jws`, so it is unsurprising that we didn't eliminate any unnecessary synchronization here. `instantdb` and `slice` are large benchmark programs that

used much of the AWT graphics library, and our context-sensitive alias analysis did not scale to these programs; using a context-insensitive alias analysis failed to catch thread-local synchronization operations.

Both reentrant lock analysis and enclosed lock analysis had a small impact on most benchmarks, but made significant contributions to a select few. For example, reentrant lock analysis in general tended to eliminate operations that were also removed by thread-local analysis; however, the `proxy` benchmark benefited from optimizing reentrant locks that were not thread-local, and thus were not optimizable with that technique. Similarly, enclosed lock analysis made an impact on `jlogo` by eliminating 12% of the dynamic synchronization operations through specializing a particular call site; these synchronization operations were not thread-local and could not have been optimized by other algorithms. There are two reasons why the enclosed and reentrant lock analyses did not make an impact on benchmarks across the board. First, the benchmarks exhibit by far more thread-local operations than enclosed and reentrant locks combined, and most cases of simple reentrant and enclosed locks had been already optimized out manually by programmers. For example, rather than use the synchronized `Vector` class to store hash table elements in their buckets, the implementors of `java.util.Hashtable` designed a custom, unsynchronized linked list class. While our analyses would have removed this source of overhead, programmers who did not have these analyses available to them did the optimizations themselves, at the cost of more complex code. All of the benefit of enclosed lock analysis in our benchmarks came from unique enclosing locks, suggesting that following unshared and immutable fields is not a useful technique for optimizing common Java programs. The second reason why the enclosed and reentrant lock analyses were not effective on all benchmarks involves inaccuracies in alias analysis. For example, all of our programs have synchronization on `OutputStreamWriters` that are enclosed by `PrintStreams`. Although our analyses identified these operations as being enclosed, our implementation was unable to optimize them; we need to optimize some `OutputStreamWriters` and not others, but we cannot use dispatch to tell the difference, because the synchronization statement is in the `BufferedWriter` class, not the `OutputStreamWriter` class. A more precise alias analysis could address this problem at the expense of compilation time.

The extent to which the reductions in dynamic synchronization operations translated into execution time speedups depended on the frequency of synchronization operations in the programs. For example, Table 2 shows that `jlex` and `jgl` do far more synchronization operations per second than the other benchmarks, and that translated into a dramatic speedup for these benchmarks. Figure 6 shows the execution speed of our optimized benchmark programs relative to the unoptimized versions. In the graph, the bars represent the execution speed improvement due to all analyses combined, relative to the

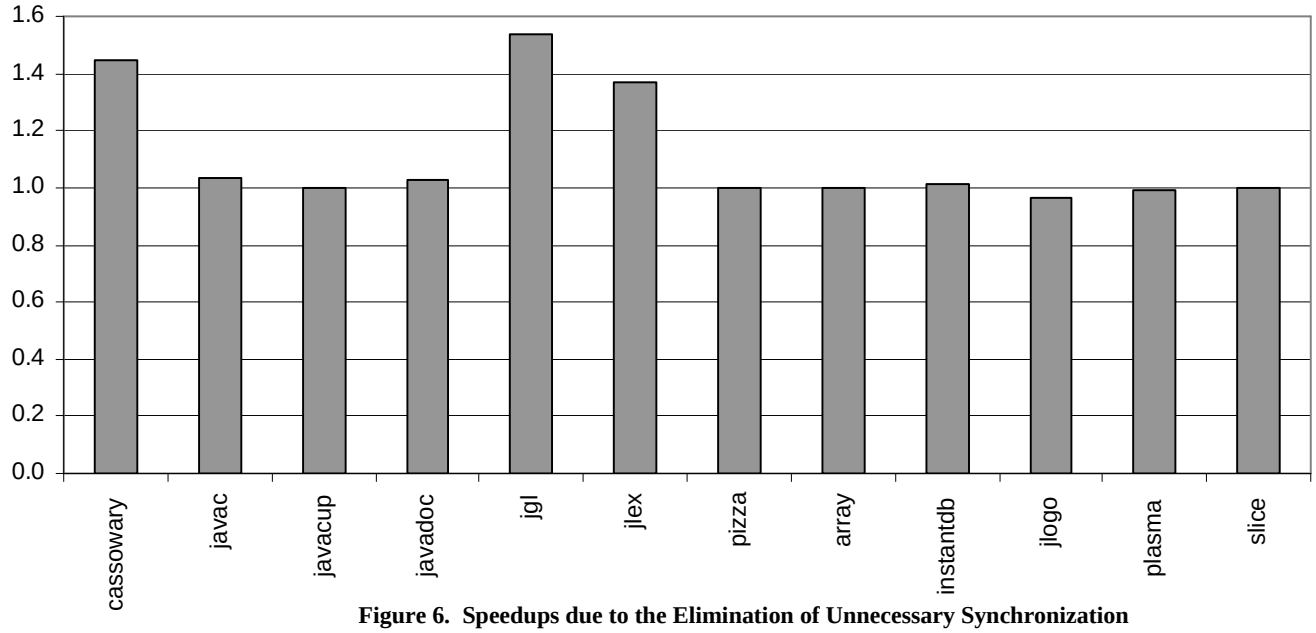


**Figure 6. Speedups due to the Elimination of Unnecessary Synchronization**

unoptimized versions. Because our analyses are particularly relevant for multithreaded benchmarks running on multiprocessors, these numbers were collected on a Solaris machine with four 90 MHz hyperSPARC processors and 250 MB of RAM. Since this machine is a few years old, and multiprocessor synchronization costs in cycles typically increase as clock speed rises, our speedup numbers are probably conservative. Our runtime platform was the JDK 1.2.103, a high-performance commercial JIT compiler with an efficient implementation of synchronization [OK99][ADG+99]. All measurements represent an average of five executions, preceded by three runs to warm up the caches. We were unable to collect meaningful execution times for the benchmarks `jws`, `proxy`, and `raytrace`.

The synchronization analyses were very effective for `cassowary`, `jgl`, and `jlex`, speeding up their execution by 37-53%. The speedups are due to the high frequency of synchronization operations, the high cost of synchronization operations in these benchmarks (particularly `cassowary`) relative to other benchmarks, combined with the effectiveness of our analyses on these programs. In other benchmarks in which our analyses eliminated a substantial proportion of the synchronization operations, such as `javacup` and `pizza`, the impact on total execution time was small, because synchronization accounted for a small portion of running time.

### 5.2 A Simpler Synchronization Model

In order to determine whether our analyses can support a simpler synchronization model, where by default all public methods of each class are synchronized, we performed two experiments. In the first, we wrote a simple program that illustrates

| benchmark | Reentrant | Non-Thread-Local Reentrant | Thread-Local | Total Statements |
|---|---|---|---|---|
| cassowary | 178 | 1 | 807 | 827 |
| javac | 136 | 1 | 1695 | 1721 |
| javacup | 199 | 1 | 835 | 854 |
| javadoc | 208 | 1 | 1305 | 1331 |
| jgl | 167 | 1 | 683 | 702 |
| pizza | 118 | 1 | 1154 | 1174 |
| array | 160 | 51 | 484 | 693 |
| jlogo | 154 | 1 | 660 | 681 |
| plasma | 318 | 318 | 0 | 1842 |
| proxy | 161 | 47 | 510 | 736 |
| raytrace | 301 | 165 | 685 | 1331 |
| slice | 324 | 324 | 0 | 1863 |

**Table 3: Synchronization Statements Optimized in the Simple Synchronization Model**

some characteristics of a web server or database server. Appendix A lists the web server code, including a simple driver application. The application has several threads that read a table data structure, and one thread that writes to it. The table is implemented as a closed hash table, where all entries with the same hash code are stored in the linked list for the corresponding hash bucket. Although most current implementations of hash tables (e.g., `java.util.Hashtable`) implement their own linked-list data structure for efficiency, we believe it is a better design to reuse an existing list class if efficiency considerations permit. As a reusable class, our list implementation included synchronization on all public methods. However, our analyses were able to eliminate all synchronization on the list, because it was enclosed by the (globally unique) hash table data structure. The resulting application sped up by 35% vs. the original unoptimized version, matching the performance of a hand-optimized version of the same benchmark. This example demonstrates that our analyses (and in particular, enclosed lock analysis) have the potential to make a cleaner programming style more practical.

In a second experiment, we modified the Java class libraries and a subset of our applications to add synchronization to all public methods. Table 3 shows the static number of synchronization points optimized by our analyses in this experiment. For most programs, thread-local analysis (shown in the fourth column) was able to eliminate virtually all of the synchronization in these programs, effectively eliminating the extraneous overhead that would be imposed by the more natural synchronization model. Because this synchronization model leads to significant reentrant synchronization, our reentrant lock analysis was able to eliminate 10-30% of the static synchronization points in these programs (second column). The role of reentrant lock

analysis was particularly important for multithreaded programs, as shown in the third column of Table 2, which lists the reentrant synchronization points that are not also thread-local. In the multithreaded benchmarks, reentrant lock analysis typically eliminates 10% of the static synchronization points in the program, in addition to what thread-local analysis is able to find. Enclosed lock analysis was unable to optimize these programs. It is likely that remaining imprecisions in the alias analysis, together with the increased use of synchronization and the inherent difficulty of identifying and optimizing enclosed locks were the causes.

Since we have not yet provided a facility to override the default of adding synchronization to every method, and our Java programs were not designed with this synchronization paradigm in mind, most programs deadlock when run with synchronization added to all public methods. However, we were able to evaluate the effect of our analyses on `javadoc`. The version of `javadoc` with all public methods synchronized executes in 52.2 seconds; in optimizing this version we were able to reduce the runtime to 37.6 seconds, which is faster than the original, manually optimized program. These results imply that our analyses are able to successfully mitigate the performance impact of a cleaner synchronization model.

### 5.3 Analysis Time

The time to perform our analyses was substantial, given that they are whole-program analyses and that our implementation is not optimized. Nevertheless, our thread-local analysis times ranged from 2 minutes for `cassowary` to 17 minutes in the case of `plasma`, running on an Sun ULTRASPARC with about 500 MB of RAM. Reentrant and enclosed lock analysis took between 2 and 27 hours, similar to the amount of time taken by alias analysis. Profile information suggests that the analysis time for reentrant and enclosed locks is not due to computation of lock information, but due to the overhead of the analysis infrastructure and an intraprocedural flow-sensitive alias analysis that must be run in conjunction with lock analysis. In a production system, an SSA representation combined with a specialized interprocedural analysis infrastructure would likely improve the performance of the lock analyses by an order of magnitude or more. Furthermore, our reentrant lock analysis can be run without the enclosed lock analysis portion, leading to significantly increased performance. Ruf's work has shown that thread-local analysis can be run with a very efficient and scalable specialized alias analysis [R00]. It is likely that reentrant and enclosed lock analyses could be designed to work with a similar specialized alias analysis; this is an important area of future work.

## 6. Future Work

Several interesting areas of future work remain. Since long analysis times is a significant drawback of our work, it would be interesting to combine our field-traversing thread-local analysis with Ruf's alias analysis [R00]. Such a combination should be as efficient as Ruf's analysis, while distinguishing between field reads and writes in order to get precision better than either work could alone. Summary and unification-based alias techniques could also be applied to our unshared field analysis and enclosing lock analysis, potentially allowing better and more efficient results. An analysis similar to unshared field analysis could be applied to unboxing objects to represent them inline, inside a container object. There may also be other forms of unnecessary synchronization that could be optimized.

Perhaps the most important area of future work is designing and evaluating more effective language mechanisms for synchronization. As described earlier, Java's default of not synchronizing without an explicit annotation makes easy to omit a single declaration, possibly leading to dangerous data races. While alternative mechanisms, such as synchronizing at every method call, have been proposed, it is not clear whether such mechanisms are useful in practice. Our work suggests that compiler technology can eliminate the runtime costs of such models, but more evaluation of their effects on software engineering is necessary.

## 7. Conclusion

The synchronization analyses described in this paper, namely thread-local, lock, and unshared field analysis, resulted in a large decrease in dynamic synchronization operations for most programs, enabling programmers to use clean synchronization models without incurring significant extra cost. The thread-local algorithm was the most effective of the three: its dramatically increased performance over previously published thread-local analyses demonstrates the importance of considering thread interactions when eliminating unnecessary synchronization from Java programs. Three of our benchmarks experienced speedups of 37-53%; other benchmarks we evaluated also improved, but to a far lesser extent, because the frequency of synchronization operations in them was low. The results show that our analyses for automatically eliminating unnecessary synchronization enable programmers to more easily write reusable, maintainable, and correct multithreaded programs without worrying about excessive synchronization cost.

## Acknowledgments

This work has been supported by a National Defense Science and Engineering Graduate Fellowship, ONR contract N00014-96-1-0402, NSF grant CCR-9503741, NSF Young Investigator Award CCR-9457767, and gifts from Sun Microsystems, IBM, Xerox PARC, and Object Technology International. We appreciate feedback from Allan Heydon, John

Whaley, Martin Rinard, Bruno Blanchet and Bill Pugh. We also thank the authors of our benchmarks: JavaSoft (javac, javadoc, jws), Philip Wadler (pizza), Andrew Appel (jlex and javacup), and Greg Badros (cassowary).

## References

[ACSE99] J. Aldrich, C. Chambers, E.G. Sirer, and S.J. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Proc. of 6th International Static Analysis Symposium*, September 1999.

[ADG+99]O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y.S. Ramakrishna, D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proc. of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[B99] B. Blanchet. Escape Analysis for Object-Oriented Languages. Application to Java. In *Proc. of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[BH99] J. Bogda and U. Holzle. Removing Unnecessary Synchronization in Java. In *Proc. of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[BKMS98]D. Bacon, R. Konuru, C. Murthy, M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proc. of SIGPLAN 1998 Conference on Programming Language Design and Implementation*, June 1998.

[C98] J. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. In *Proc. 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.

[CGS+99]J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Proc. of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proc. of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1996.

[DLNS98]D.L. Detlefs, K. Rustan M. Leino, G. Nelson, and J.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*: 49(2), March 1998.

[H92] S.P. Harbison, “Modula-3”, Prentice Hall, 1992.

[HN99] A. Heydon and M. Najork. Performance Limitations of the Java Core Libraries. In *Proc. of the 1999 ACM Java Grande Conference*, June 1999.

[FKR+98]R. Fitzgerald, T.B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an Optimizing Compiler for Java. Microsoft Technical Report, November 1998.

[GDDC97]D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. In *Proc. of the 12th Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1997.

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

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

[LR80] B. Lampson and D. Redell. Experience with Processes and Monitors in Mesa. In *Communications of the ACM*: 23(2), February 1980.

[M99] David McAllester. On the Complexity Analysis of Static Analyses. In *Proc. of 6th International Static Analysis Symposium*, September 1999.

[OK99] T. Onodera and K. Kawachiya. A Study of Locking Objects with Bimodal Fields. In *Proc. of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[P96] J. Plevyak. Optimization of Object-oriented and Concurrent Programs. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1996.

[PZC95] J. Plevyak, X. Zhang, and A. Chien. Obtaining Sequential Efficiency for Concurrent Object-Oriented Languages. In *Proc. of 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995.

[P99] W. Pugh. Fixing the Java Memory Model. In *Proc. of Java Grande Conference*, June 1999.

[R00] Eric Ruf. Effective Synchronization Removal for Java. In *Proc. of SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[RR99] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proc. of SIGPLAN 1999 Conference on Programming Language Design and Implementation*, May 1999.

[SGGB99] E. G. Sirer, R. Grimm, A. J. Gregory and B. N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proc. of 17th ACM Symposium on Operating Systems Principles*, December 1999.

[S88] Olin Shivers. Control-Flow analysis in Scheme. In *Proc. of SIGPLAN 1988 Conference on Programming Language Design and Implementation*, July 1988

## Appendix A: Webserver Code

```
import java.util.Random;

class Pair {
  private Object first;
  private Object second;

  public Pair(Object f, Object s) {
    first = f; second = s; }

  public synchronized Object getFirst() {
    return first; }
  public synchronized Object getSecond() {
    return second; }
  public synchronized void setFirst(Object f)
    { first = f; }
  public synchronized void setSecond(
         Object s) { second = s; }
}

class Table {
  private List entries[];
  private int capacity;

  public Table() {
    capacity = 13587;
    entries = new List[capacity];
    for (int i = 0; i < capacity; ++i)
      entries[i] = new List();
  }

  public synchronized Object get(Object key) {
    return getEntry(key).getSecond();
  }

  public synchronized void put(Object key,
         Object value) {
    Pair entry = getEntry(key);
    entry.setSecond(value);
  }

  private synchronized Pair getEntry(Object
         key) {
    int index = key.hashCode() % capacity;
    List l = entries[index];
    l.reset();
    while (l.hasMore()) {
      Pair p = (Pair) l.getNext();
      if (p.getFirst().equals(key))
        return p;
    }
    Pair p = new Pair(key, null);
    l.add(p);
    return p;
  }
}

class List {
  private Pair first;
  private Pair current;

  public synchronized void reset() {
    current = first; }
  public synchronized boolean hasMore() {
    return current != null; }
  public synchronized Object getNext() {
    if (current != null) {
      Object value = current.getFirst();
      current = (Pair) current.getSecond();
      return value;
    }
    else
      return null;
  }

  public synchronized void add(Object o) {
    first = new Pair(o, first); }
}

class WriterThread extends Thread {
  public void run() {
    int myMaxNumber = 100;
    while (myMaxNumber < 10000) {
      for (int i = 0; i < 100; ++i) {
        Webserver.dataTable.put(
          new Integer(myMaxNumber),
          String.valueOf(myMaxNumber));
        myMaxNumber++;
      }
      synchronized(Webserver.maxNumberLock) {
        Webserver.maxNumber = myMaxNumber;
      }
    }
    System.out.println("Writer complete");
  }
}

class ReaderThread extends Thread {
  public void run() {
    int myMaxNumber;
    Random rand = new Random();
    for (int i = 0; i < 1000; ++i) {
      synchronized(Webserver.maxNumberLock) {
        myMaxNumber = Webserver.maxNumber;
      }
      for (int j = 0; j < 100; ++j) {
        int index = Math.abs(
            rand.nextInt()) % myMaxNumber;
        Webserver.dataTable.get(
            new Integer(index));
      }
    }
    System.out.println("Reader complete");
  }
}

public class Webserver {
  public static void main(String args[]) {
    /* set up data table */
    maxNumber = 100;
    dataTable = new Table();
    maxNumberLock = new Object();
    for (maxNumber = 0; maxNumber < 100;
         ++maxNumber) {
      dataTable.put(new Integer(maxNumber),
        String.valueOf(maxNumber));
    }
    for (int threadNum = 0; threadNum < 8;
         ++threadNum) {
      new ReaderThread().start();
    }
    new WriterThread().start();
  }
  public static Table dataTable;
  public static int maxNumber;
  public static Object maxNumberLock;
}
```