Critical Section, Be Gone!

Jeff Bogda and Ambuj Singh
Department of Computer Science
University of California
Santa Barbara, CA 93106
{bogda,ambuj}@cs.ucsb.edu

Abstract

To remove the costly synchronization from Java programs, many researchers employ an escape analysis that identifies objects that cannot be accessed by multiple threads and hence do not require synchronization. This approach, however, does not attempt to optimize objects that *do* escape a thread. These escaping objects may be optimized when, generally speaking, no thread writes to them after they escape their creator thread.

We present a shape analysis that constructs an accurate static picture of the connectivity of shared objects. Its results feed a flow-sensitive data-flow analysis that determines which objects a thread has locked when it writes to a location visible to other threads. If an object's lock does not protect such a write, the object does not need to be locked. Ignoring the locking operations of this object will not only reduce the amount of locking but also increase program concurrency, potentially leading to a significant speed-up.

1. Introduction

Writing multi-threaded programs has always been, to say the least, tricky and error-prone, but Java's **synchronized** keyword attempts to make the programming easier. The keyword enables a programmer to specify succinctly that a thread must have exclusive access to certain shared variables and objects. It defines a block of code a thread should execute atomically, requiring the run-time system to obtain a lock upon entering the region and to release the lock upon exiting it.

The Java programming language supports two ways to specify a critical region: assign a method the **synchronized** modifier or define an explicit synchronized block of code. The first option conveniently specifies that the entire body of the method should serve as a critical section. In the case of a synchronized instance method, the protecting object, or *guard*, is the receiver, while in the case of a synchronized static method, the guard is the global class object representing the class in which the method appears. Alternatively, with a synchronized block, which translates to a monitorenter and a number of monitorexit bytecodes, the programmer can specify explicitly which object the run-time system should lock. We refer the reader to [13] or [17] for a more elaborate explanation of Java syntax.

Because of the ease of specifying critical sections, multi-threaded programs and, more importantly, multi-threaded libraries are becoming more widespread. Unfortunately the cost of acquiring and releasing a lock is non-negligible [7,14], prompting seriously concerned programmers to avoid critical sections whenever possible. Striving to speed up their applications, they often avoid synchronized library classes entirely or write complicated hand-tuned, thread-safe data structures.

To discourage this sidestepping, recent work has looked at ways to reduce the locking overhead. In one corner, researchers suggest that the run-time system can perform the lock operations more efficiently [1,4,18]. In another, researchers propose that an analysis determine points in a program where lock operations are superfluous [2,5,7,8,21]. These two approaches, both shown to be effective, are completely orthogonal.

In the latter stream, two types of static, whole-program analyses have emerged. One [7,8,21] constructs a static picture of the heap in order to predict that certain run-time objects will not possibly be visible to more than one thread. Researchers describe these objects as *thread-local*, or *thread-private*. The run-time system can safely ignore locking of these objects. It continues to lock objects visible to multiple threads, or *thread-escaping* objects. The other type of analysis [2,10] statically tracks a program's pattern of locking in order to eliminate redundant locking. For example, if it finds that a thread always holds lock A when it acquires lock B, then the acquisition of lock B is unnecessary. These two types of analyses eliminate a significant amount of synchronization, but they do no eliminate *all* unnecessary synchronization. The work behind this paper differentiates code that writes to memory and code that reads from memory in order to eliminate more synchronization.

A critical section only needs to protect a shared object or variable to which threads *write*. Two threads can simultaneously read data, but a thread should not write to it without proper synchronization. The lock of a thread-escaping object serves no purpose if no critical section that the object protects writes to shared data. The run-time system does not need to lock these objects, thereby increasing concurrency and decreasing the number of lock operations.

We propose a program analysis (section 4) that identifies and optimizes these objects. To do so, it recognizes writes to memory locations visible to more than one thread and identifies objects that are locked when these writes occur. Because of the former task, we label the analysis a *visible-write analysis*.

To recognize writes to shared data, the visible-write analysis seeks places where the program writes to a static field or to a thread-escaping object, where a shape analysis (section 3) identifies the thread-escaping objects. The shape analysis presented in this paper exceeds previous shape analyses by capturing the structure of thread-escaping objects, in addition to thread-local objects. While identifying writes to shared data, the visible-write analysis maintains a collection of objects that are currently locked. Since Java programs acquire and release locks in a stack-like fashion, the visible-write analysis needs to be flow-sensitive in order to match an object to the critical sections it protects. Exploiting the flow-sensitivity, it uses the extra knowledge of *when* objects escape the thread to make the identification of visible writes more precise.

```
public class Matrix {
   private Object[][] cells;

   public Matrix( int dim ) {
      cells = new Object[dim][dim];
   }

   public synchronized Object readCell( int i, int j ) {
      return cells[i][j];
   }

   public synchronized Matrix subMatrix( int startRow,
         int startCol, int dim ) {
      Matrix sub = new Matrix( dim );

      for( int i=0; i<dim; i++ )
         for( int j=0; j<dim; j++ )
            sub.writeCell( i, j, readCell( i+startRow, j+startCol ));

      return sub;
   }

   public synchronized void writeCell( int i, int j, Object value ) {
S1:   cells[i][j] = value;
   }
}

public class MatrixMultiply extends Thread
{
   private int iOffset;
   private int jOffset;
   private int bSize;
   private int q;
   private Matrix A;
   private Matrix B;
   private Matrix C;

   public MatrixMultiply( int iOffset, int jOffset, int bSize, int q,
         Matrix A, Matrix B, Matrix C ) {
      this.iOffset = iOffset;
      this.jOffset = jOffset;
      this.bSize = bSize;
      this.q = q;
      this.A = A;
      this.B = B;
      this.C = C;
   }

   // performs op1 = op1 + op2
   protected void add( Matrix op1, Matrix op2 ) {
      for( int i=0; i<bSize; i++ )
         for( int j=0; j<bSize; j++ ) {
            int sum = ((Integer)op1.readCell( i, j )).intValue() +
                     ((Integer)op2.readCell( i, j )).intValue();
            op1.writeCell( i, j, new Integer( sum ));
         }
   }

   public static void main( String[] args ) {
      final int dim = 100;

      Matrix A = new Matrix( dim );
      Matrix B = new Matrix( dim );
S2:   Matrix C = new Matrix( dim );

      for( int i=0; i<dim; i++ )
         for( int j=0; j<dim; j++ ) {
S3:         A.writeCell( i, j, new Integer( i+j ));// initialize to any value
S4:         B.writeCell( i, j, new Integer( i ));   // initialize to any value
S5:         C.writeCell( i, j, new Integer( 0 )); // initialize to 0
         }

      int bSize = 10;
      for( int i=0; i<dim/bSize; i++ )
         for( int j=0; j<dim/bSize; j++ )
S6:         new MatrixMultiply( i*bSize, j*bSize, bSize, dim/bSize,
                  A, B, C ).start();

      // wait until threads finish
      while( Thread.activeCount()>1 ) { }
   }

   // returns m1 x m2
   protected Matrix mm( Matrix m1, Matrix m2) {
      Matrix soln = new Matrix( bSize );

      for( int i=0; i<bSize; i++ )
         for( int j=0; j<bSize; j++ ) {
            int val = 0;
            for( int k=0; k<bSize; k++ )
               val = val + ((Integer)m1.readCell( i, k )).intValue() *
                  ((Integer)m2.readCell( k, j )).intValue();
            soln.writeCell( i, j, new Integer( val ));
         }
      return soln;
   }

   public void run() {
S7:   Matrix temp = new Matrix( bSize );

      for( int i=0; i<bSize; i++ )
         for( int j=0; j<bSize; j++ )
            temp.writeCell( i, j, new Integer( 0 ));// initialize to 0

      for( int k=0; k<q; k++ ) {
S8:      Matrix blockOfA = A.subMatrix( iOffset, bSize*k, bSize );
S9:      Matrix blockOfB = B.subMatrix( bSize*k, jOffset, bSize );
S10:     add( temp, mm( blockOfA, blockOfB ));
      }
      // copy temp to result matrix C
      for( int i=0; i<bSize; i++ )
         for( int j=0; j<bSize; j++ )
S11:        C.writeCell( i+iOffset, j+jOffset, temp.readCell( i, j ));
   }
}
```

Figure 1. Sample multi-threaded matrix multiplication program that uses a generic, thread-safe data structure.

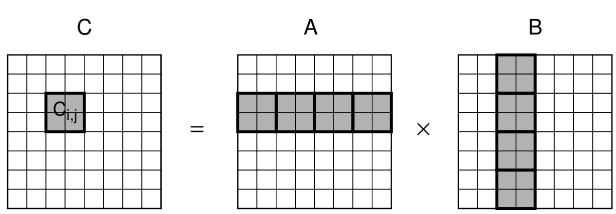
The visible-write analysis hands the optimizer a set of thread-escaping objects that do not protect a visible write. The optimizer in turn transforms the program (section 5) in such a way that it never acquires the locks of these objects, thereby eliminating critical sections. The decrease in the number of lock operations and the potential increase in concurrency may speed up multi-threaded applications (section 6).

2. Example

Frequently an object's lock serves only to protect access to the object itself. In these cases, this optimization will be most effective if a program contains thread-escaping objects that are only read. We believe that typical Java programs use such objects. Programs tend to store many global, immutable objects in static fields. For example, at start-up the core classes establish global java.util.Locale objects that subsequently go unchanged. Furthermore, many parallel algorithms, such as the canonical Traveling Salesman Problem, share data structures globally but never update them. In this section we present an implementation of matrix multiplication that also shares read-only data structures. We will use this example throughout the paper to demonstrate the analyses.

Consider the sample multi-threaded matrix multiplication program in Figure 1, which follows the block matrix multiplication algorithm described in [16]. After constructing and initializing three 100-by-100 matrices—A, B, and C—it spawns 100 threads to calculate in parallel $C = A \times B$. Each thread is responsible for calculating a distinct 10-by-10 block in C by reading, multiplying, and summing the appropriate blocks of A and B. Each thread performs the simple algorithm in Figure 2. The main routine terminates only after each spawned thread has finished. For readability, main tests this condition with a spin loop on the number of active threads. In practice, the program would instead invoke join on the spawned threads in order to save CPU cycles.

Another programmer could have written this program with two-dimensional integer arrays and explicit synchronized blocks to control reading and writing of these arrays. His rationale presumably would have been to speed up execution; however, this comes at the cost of both increased code complexity and loss of object-orientation. Instead, we have utilized a generic square Matrix class, which enhances readability and reusability and provides thread-safe access. This approach improves readability and reusability by allowing matrices to share code and to hold values of any reference type. More importantly, in multi-threaded usage every instance of the class can withstand concurrent access



$$C_{i,j} = \sum_{k=0}^{\frac{dim}{bSize}-1} A_{i,k} \times B_{k,j}$$

Figure 2. Calculation of a block in the blocked matrix multiplication problem.

since the bodies of all methods are critical sections protected by the receiver. This approach—arguably consistent with the flavor of Java—lets the programmer focus on the parallel algorithm, while leaving the efficiency matters to the compiler and run-time system.

How might a compiler optimize this program? By observing the use of objects, it can recognize that the matrices created at statement S7, as well as the matrices returned by the calls to subMatrix and mm at S8, S9, and S10, are used locally (that is, only by the executing thread) to calculate the block of C. It can then inform the run-time system not to synchronize these objects because two threads will never contend for their locks. The ideas behind [7,8,21] address this optimization.

On the other hand, all threads share and access the matrices A, B, and C. The main thread first initializes them and then passes them to each thread object. No thread subsequently writes to A and B—only to C. Recognizing this, the compiler can inform the run-time system not to lock A and B. Threads can safely invoke their methods concurrently.

The next three sections present an optimization that automatically eliminates unneeded synchronization operations. Our shape analysis constructs a static picture of A, B, and C, before our visible-write analysis determines that the locks of A and B do not protect any visible writes. Last, our program optimization transforms MatrixMultiply so that it never acquires A or B's lock.

3. Shape Analysis

The goals of the shape analysis are to capture statically the structure of run-time objects, to relate them to program points, and to identify which objects are thread-escaping. This information enables a subsequent data-flow analysis to determine which objects guard critical sections and which writes may be visible to another thread.

3.1 Assumptions

When describing the analysis, we use the following notation.[1]

x = y	assignment
x.f = y	store into an instance field
y = x.f	load from an instance field
C.f = y	store into a static field
y = C.f	load from a static field
foo(a_0,...,a_n)	method invocation
monitorenter x	acquisition of x's lock
monitorexit	release of currently held lock

In order to treat an arbitrary Java program in this form, we make the following assumptions. First, all array objects have a single instance field named array.[2] Thus we do not distinguish array cells. Second, instance methods take the receiver (a_0) as their first argument (p_0). Third, all methods take the destination variable (a_{n-1}) as the second-to-last argument (p_{n-1}) and the thrown exception (a_n) as the last argument (p_n). Thus, conceptually a method returns or throws an object by passing it back as a

[1] Since the shape analysis approximates the structure of heap *objects*, it can safely ignore statements involving primitive types. The data-flow analysis of the next section, however, cannot ignore such statements because it must survey all writes the program makes.

[2] Note that n-dimensional arrays have n-1 nested subarrays, each with an array field.

Notation	Explanation
a, b, c, d, x, y ∈ V	Program variables
f, g ∈ F	Field names
m, foo′ ∈ M	Methods
C_0, C_1, ... ∈ C	Object contexts
context : (V.F*) → C	Returns the object context representing a given variable and path, where the path is a sequence of field names specifying successive pointer dereferences. If a context does not exist, one is created.
fields : V → 2^F	Returns the set of reference field names of the class (and its subclasses) defining a given variable.
shared : C → { true, false}	True if an object represented by the specified context may be reachable from a static field; false, otherwise.
methodsInvoked : M → 2^M	Returns the set of method implementations that satisfy the signature of a given method. We use a resolution scheme similar to rapid type analysis (RTA) [3] to identify target methods.
re-entry : C × M → 2^C	Returns the set of contexts of a given method that a given context may flow into as a result of program recursion.
reference : V → { true, false }	True if a given variable is of type reference; false, otherwise.
reachable : C → 2^C	Returns all contexts (including itself) reachable from a given context in the shape graph.
written-to : C → { true, false }	True if the program writes to an object represented by the given context; false, otherwise.
protective : C → { true, false }	True if a visible write occurs while a given context is locked, else false.
locks : C → 2^C	Denotes the non-thread-local contexts whose locks are held when the program writes to a given context.
writes : M → { true, false }	True if a given method (or any of its callees) executes a visible write.
map : C → C	Maps a callee's context to a caller's. If the given context is *shared*, returns the same context; otherwise, returns the corresponding context reachable from the actual parameters.
$\text{escaped}_{IN} \in 2^C$	The set of contexts that may have escaped the thread *before* a given instruction executes.
$\text{escaped}_{OUT} \in 2^C$	The set of contexts that may have escaped the thread *after* a given instruction executes.
$\text{escaped}_{GEN} \in 2^C$	The set of contexts that may escape the thread as a result of a given instruction.
$\text{guards}_{IN} \in 2^C$	The stack of contexts for which the program may have acquired their locks *before* a given instruction executes. The most recently acquired lock is on top.
$\text{guards}_{OUT} \in 2^C$	The stack of contexts for which the program may have acquired their locks *after* a given instruction executes.

Table 1. Notation and predicates.

parameter.[1] For instance, the Java statement throw x is equivalent to p_n = x, and the statement return y is equivalent to p_{n-1} = y.

A native method complicates the analysis because its code is not available. This forces the analysis to make assumptions about the method's behavior. If a native method is a known Java Development Kit (JDK) library routine (approximately 100 such methods in the programs analyzed), we assume that it behaves according to its specification and simulate it. Consider the native method start of java.lang.Thread. It places its receiver in a global pool of threads and causes the run-time system to spawn a new thread, which begins executing the run method of an object implementing the Runnable interface. On the other hand, we try to handle an unfamiliar native method conservatively and assume the worst behavior; however, we assume that it does not reenter the Java program before returning.

3.2 Constraints

We employ a context-sensitive, flow-insensitive interprocedural program analysis to approximate the heap structure. It calculates the strongly connected components of the static call graph and then processes the components in reverse topological order, choosing the method ordering within a component arbitrarily. It studies each method independently of its calling context(s) and constructs an appropriate method summary. It only analyzes a method once, except in the case of program recursion, in which it iterates over the component until it reaches a fixed point. Since the analysis is context-sensitive, it differentiates all call sites within the program. Since it is flow-insensitive, the analysis does not maintain a picture of the heap at each program point.

Our implementation uses a graph to capture the structure of heap objects. A node in the graph represents an *object context*, which statically encapsulates a unique set of run-time objects that flow through a given method. In the remainder of the paper we abuse the terminology and use *context* synonymously with *object*; however, the distinction is grossly important. An object that flows to more than one method often maps to several contexts, and a context often corresponds to several objects. Two nodes C_i and C_j are connected by a directed edge labeled f if an object represented by C_j is accessible via a pointer dereference of field f of an object represented by C_i. The authors of [8] use the term *connection graph* to label this abstraction, while the authors of [21] use *points-to escape graph*. We will use the generic term *shape graph*, which was recently used in [22]. Other notation, convenience functions, and predicates appear in Table 1. The second half of the table applies only to the visible-write analysis of section 4. Note that the *shared* attribute denotes a context representing a thread-escaping object.

Initially, the analysis associates each variable, including formal parameters, and each static field to a unique context. It marks the contexts representing static fields as *shared*, signaling that the objects referenced by these fields are thread-escaping. This is by definition true because any thread may use the getstatic instruction to access these objects. As the analysis examines a statement in the program, it determines if the instruction causes any objects to escape the thread and if it potentially causes two paths of pointer dereferences to refer to the same object.

[1] We abuse the semantics of Java here and treat p_{n-1} and p_n as if they were pass-by-reference. This simplifies the presentation and does not adversely affect the analysis.

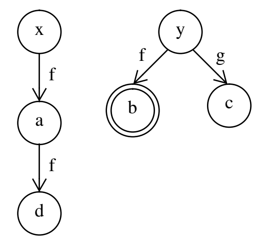
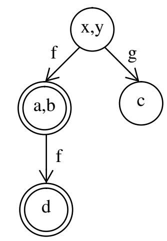
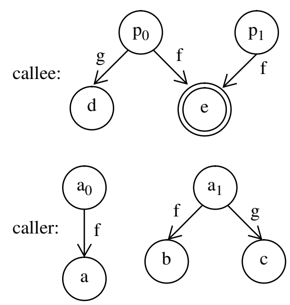
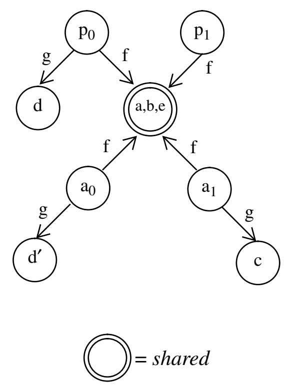
Instruction	**Constraints**	**Before Instruction**	**After Instruction**
x = y Similarly, x.f = y y = x.f C.f = y y = C.f	We recursively equate the contexts corresponding to x and y and maintain the invariant that all contexts reachable from a *shared* context are *shared*. Let the operand ≡ denote the following: context(x) = context(y) ∀ f ∈ fields(x) ∪ fields(y): context(x.f) ≡ context(y.f) if shared(context(x)): shared(context(x.f)) = T		
foo(a_0,...,a_n)	We replicate in the caller converging paths present in a callee's view of the shape graph. If a path does not exist in the caller, we create one. Furthermore, if a context in a callee is marked *shared*, the caller shares this context with the callee. In the case of recursion, we collapse the cycle of recursive contexts into a single context. ∀ foo′ ∈ methodsInvoked(foo): ∀ i, j ∈ [0..n], paths φ_i, φ_j such that context($p_i.\varphi_i$) = context($p_j.\varphi_j$): context($a_i.\varphi_i$) ≡ context($a_j.\varphi_j$) ∀ i ∈ [0..n], paths φ_i such that shared(context($p_i.\varphi_i$)): context($a_i.\varphi_i$) ≡ context($p_i.\varphi_i$) ∀ i ∈ [0..n], paths φ_i such that re-entry(context($p_i.\varphi_i$),m) ≠ ∅: context($a_i.\varphi_i$) ≡ context($p_i.\varphi_i$) ∀ C_j ∈ re-entry(context($p_i.\varphi_i$)): context($a_i.\varphi_i$) ≡ C_j		

Table 2. Constraints for the shape analysis.

The rules guiding the shape analysis of method m appear as constraints in Table 2. The table's second column gives the conditions that must hold on the shape graph if the analysis encounters the instruction in the first column. The third and fourth columns, respectively, present snapshots of a sample graph before and after applying the corresponding constraints. Every circle denotes a context, and a double circle denotes a shared context. For convenience we label a node with the variables that point to objects it embodies.

All forms of an assignment statement cause the analysis to merge the context representing the left-hand side and the context representing the right-hand side. For example, the assignment x = y causes two variables to reference the same object, so the analysis merges the object contexts representing x and y as well as the subgraphs rooted at these contexts. Ignoring the direction of the assignment mirrors Steensgaard's unification-based analyses [20].[1] In the diagram the contexts for x and y as well as for a and b become one. Furthermore, when the analysis merges subgraphs, it maintains the invariant that all contexts reachable from a *shared* context are *shared*. Note that d's context is *shared* now that a's context is *shared*.

At a method invocation the analysis reflects in the caller the *shared* attributes and connectivity of the shape graph as seen by any callee. The analysis replicates the heap structure unless the objects are thread-escaping, in which case the analysis reuses a callee's context. For example, in the diagram the actual arguments a_0 and a_1 map to the formal parameters p_0 and p_1. Since the paths p_0.f and p_1.f in the callee refer to the same context, the analysis merges the caller's contexts denoting a_0.f and a_1.f. Since the context of p_0.f is *shared*, the caller shares it with the callee. The analysis creates an unshared context for a_0.g because the context of p_0.g exists.

To handle program recursion, we represent an object that potentially re-enters a method with a single context, instead of replicating it at each call site along the recursive path. In other words, we unify all contexts along paths of recursion. For example, suppose static method m invokes itself, passing an object referenced by variable a_0. Since this object may recursively enter m, the analysis merges the context of p_0, the context of a_0, and any contexts along the path from the call site to m. In this tiny example, the recursion is direct, so there are no intermediate contexts to unify.

All methods that know that an object is thread-escaping manipulate the same context. Consequently, a change to its structure in one method automatically adjusts all other methods' views of it. This allows us to capture the union of all threads' views of thread-escaping objects and safely detect when two global variables cannot reference the same object. Regardless of the actual thread interleaving, if two statements refer to different *shared* contexts, they will refer to different objects at run-time.

3.3 Example

Table 3 shows the portions of the final shape graph involving the methods writeCell, run, and main. As we summarize how the analysis arrived at these subgraphs, we refer to the key contexts with the labels to the left of their nodes.

[1] A more precise analysis would recognize that the contexts pointed to by the left-hand side are killed by an assignment statement.

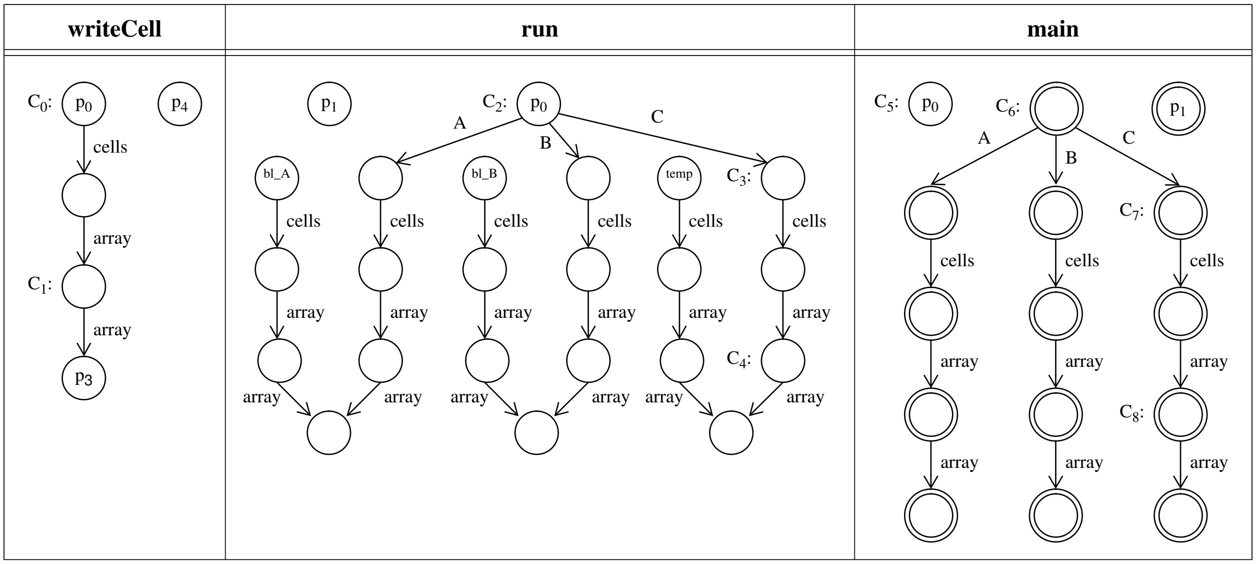


Table 3. Structure of heap objects as seen by the writeCell, run, and main methods.

Before analyzing writeCell, the analysis establishes contexts for the formal parameters of reference type (p_0, p_3, p_4). It then examines the code and discovers the store of the value argument (p_3) in the two-dimensional array nested in the receiver (p_0). Hence it applies the constraints for an assignment and merges p_3's context with the context reachable via edge array from context C_1. The method does not know anything about the structure of any object it throws (p_4).

The structure of the heap as seen by run is more extensive since it references six distinct matrices: A, B, C, temp, blockOfA, and blockOfB. Since the analysis cannot distinguish matrices of different loop iterations, these contexts potentially embody several run-time objects. The analysis learns the structures of these matrices from the calls to subMatrix, add, readCell, and writeCell. For example, the call to writeCell at line S11 causes the analysis to attach the result of the call to readCell to context C_4 via edge array. Note that the analysis does not mark any context *shared*, even though matrices A, B, and C turn out to be thread-escaping. The method only knows the properties and shape of its objects from the point they enter the method to the point they leave. It knows nothing about the situation in which it is called. Hence the program can invoke run with thread-local objects as well as with thread-escaping objects.

Because the native method start invokes the run method, the analysis copies the subgraphs reachable from the contexts of run's formal parameters (p_0,p_1) to start. Since start spawns a new thread, we treat it as if it stores its arguments into unspecified static fields. Thus, these copied structures become *shared*.

When the analysis examines the call to start at statement S6, it sees that the contexts of start's formal parameters escape the thread. Therefore, instead of copying the structure to main it unifies the contexts of the actual arguments and the contexts of start's formal arguments. This allows any changes to the structure of A, B, and C to be reflected automatically in both main and start.

4. Visible-Write Analysis

How can a program analysis recognize that matrices A and B, which are only read by multiple threads, do not require synchronized methods, while matrix C, which gets updated, does? The information provided by the shape analysis—namely that A, B, and C are potentially shared objects—is not enough.

At lines S3, S4, and S5, the program updates matrices A, B, and C. We can safely ignore the writes occurring here because the matrices have not yet escaped the thread. The run-time system places the only reference to a freshly allocated matrix onto the stack of the currently executing thread. Until that thread places the reference in a heap location visible to another thread, the object remains thread-local. Only a store into a static field or a store into another escaping object makes the object escape the thread. From the time the matrices are created until they are initialized, the program does not store their references into a shared heap location. Therefore, A, B, and C do not need to be locked during initialization.

Since we ignore writes to shared objects that have not yet escaped the thread, the visible-write analysis must have a notion of timing, which was not present in the flow-insensitive shape analysis. Thus we use a flow-sensitive data-flow analysis in order to recognize the points at which shared objects escape.

The analysis can see that the program does not write to A or to B after the matrices have escaped the thread; however, this still is not sufficient to be able to disregard all lock operations performed on A and B. Their locks may protect other shared variables and objects from access. Thus the analysis must be aware of a write to *any* thread-escaping object, not just to the one it wishes to optimize. We follow the notation of [6] and label a write to a shared memory location a *visible write*. Specifically, a visible write occurs at a write to a static field or a write to a shared object that has already escaped the thread.

Now we can state the objective of the visible-write analysis: To recognize the objects that a thread has locked when it performs a visible write. The data-flow equations given in the next section allow the analysis to detect when objects escape the thread and which objects are locked when the program writes to memory. The identification of locked objects when a visible write occurs turns

Instruction	Effect on shape graph	Effect on escaped set	Effect on guards stack
x.f = y	noteWrite(x,guards_{IN})	if reference(y) $\wedge$ context(x) $\in$ $\mathrm{escaped}_{IN}$ $\mathrm{escaped}_{GEN}$ = reachable(context(y))	
y = x.f		if reference(y) $\wedge$ context(x) $\in$ $\mathrm{escaped}_{IN}$ $\mathrm{escaped}_{GEN}$ = reachable(context(y))	
C.f = y	noteVisibleWrite(guards_{IN})	if reference(y): $\mathrm{escaped}_{GEN}$ = reachable(context(y))	
y = C.f		if reference(y): $\mathrm{escaped}_{GEN}$ = reachable(context(y))	
foo(a_0,...,a_n)	$\forall$ foo′ $\in$ methodsInvoked(foo): if writes(foo′): noteVisibleWrite({$a_i.\varphi$ | protective(context($p_i.\varphi$))} $\cup$ guards_{IN}) $\forall$ i, φ such that written-to(context($p_i.\varphi$)): noteWrite($a_i.\varphi$,{map(s) | s $\in$ locks(context($p_i.\varphi$))} $\cup$ guards_{IN})	$\forall$ foo′ $\in$ methodsInvoked(foo): $\forall$ i $\in$ [0..n], paths φ such that shared(context($p_i.\varphi$)): $\mathrm{escaped}_{GEN}$ = reachable(context($a_i.\varphi$))	
monitorenter x			guards_{OUT} = guards_{IN}.push(context(x))
monitorexit			guards_{OUT} = guards_{IN}.pop()
Initially for m(p_0,...,p_q)		$\forall$ i $\in$ [0..q-2]: // not the returned or thrown arguments $\mathrm{escaped}_{IN}$ = reachable(context(p_i))	if m is a synchronized instance method: guards_{IN}.push(context(p_0))

Table 4. Data-flow Equations for the visible-write analysis.

noteWrite(x $\in$ V, sync $\in$ 2^C)	if context(x) $\in$ $\mathrm{escaped}_{IN}$ $\wedge$ shared(context(x)):
	noteVisibleWrite(sync)
	if context(x) $\in$ $\mathrm{escaped}_{IN}$ $\wedge$ $\neg$shared(context(x)):
	noteUnknownWrite(x,sync)

noteVisibleWrite(sync $\in$ 2^C)	writes(m) = T // m is the method being analyzed
	$\forall$ s $\in$ sync $\cap$ $\mathrm{escaped}_{IN}$
	protective(context(s)) = T

noteUnknownWrite(x $\in$ V, sync $\in$ 2^C)	written-to(context(x)) = T locks(context(x)) = locks(context(x)) $\cup$ { s | s $\in$ sync $\cap$ $\mathrm{escaped}_{IN}$ }

Table 5. Helper functions augmenting the data-flow equations.

out to be non-trivial since, at the point a write occurs, the analysis does not always know the state of the object receiving the write.

4.1 Data-Flow Equations

We carry out the visible-write analysis as a flow-sensitive data-flow analysis. We make two assumptions about the programs the analysis examines. First, monitorenter and monitorexit instructions appear in a stack-like fashion such that each monitorexit corresponds to a single monitorenter. Since the Java language requires such a stack-like nesting, all bytecode generated by a correct Java-to-bytecode compiler will exhibit this structure. Second, all accesses to shared memory locations are properly synchronized. We do not want the visible-write analysis to handle all the intricacies of data races until the language designers fix the currently broken Java Memory Model. In the meantime, a programmer can use data-race detection tools to help identify improper synchronization.

At each program point the analysis carries two important pieces of information: the set of objects that more than one thread can access and the stack of currently locked objects. We let $\mathrm{escaped}_{IN}$ denote the escaped contexts immediately before an instruction executes, $\mathrm{escaped}_{OUT}$ denote those immediately after an instruction executes, and $\mathrm{escaped}_{GEN}$ denote those that escape the thread as a result of the instruction. The general data-flow equation

$$\mathrm{escaped}_{OUT} = \mathrm{escaped}_{IN} \cup \mathrm{escaped}_{GEN}$$

holds. Note that once a context escapes the thread, it has forever escaped; no KILL set appears in the equation. Additionally, since an invoked method does not know the state of its incoming formal parameters and objects reachable from these parameters, we temporarily assume they have already escaped and include their contexts in the escaped set. Thus the escaped set holds a subset of the shared contexts as well as the contexts entering the method being analyzed.

We let guards_{IN} denote the stack of object contexts known to be locked before an instruction executes and guards_{OUT} denote the stack after an instruction executes. Since this is a stack, the only valid operations on it are push and pop. The former places a context on the top of the stack, and the latter returns the stack without the topmost context. When the analysis examines a synchronized instance method, the stack initially holds the receiver context.

In addition to this data-flow information, the analysis attaches descriptive properties to nodes of the shape graph. It marks a context *written-to* if the program writes to an object denoted by this context. Also, the set *locks* captures the objects a thread currently has locked when the thread writes to the object. The *protective* attribute signals that a thread holds this context's lock when a visible write occurs. Finally, the analysis marks a method as *writes* if the method, or one of its callees, executes a visible write. At the point a write occurs, the analysis generally does not know if another thread can see the result. For example, when the program calls writeCell on C at line S11, the write is indeed a visible write, but when the program calls writeCell on temp, the write is not visible to another thread. The analysis uses these properties to propagate up the call chain the fact that a write occurs. Eventually a caller will know if the write is a visible write.

The data-flow equations appear in Table 4, and some auxiliary functions appear in Table 5. The first column of Table 4 lists the instructions important to the analysis. The second column describes the necessary changes to attributes of shape graph nodes as a result of analyzing the instruction. The third column presents the changes to the escaped set, while the rightmost column presents the changes to the stack of protecting objects.

Read instructions cause the analysis to add contexts to the escaped set. When it encounters a read of a static field (y = C.f), it notes that all contexts reachable from y's context have escaped the thread. When it encounters a read of an instance field (y = x.f), it adds the context of y to the escaped set if x's context has escaped.

Write instructions cause the analysis not only to update the escaped set but also to modify the properties of shape graph nodes. When the program writes to a static field (C.f = y), a visible write unquestionably occurs. Accordingly, the analysis marks the method being analyzed as *writes* and records that all contexts whose locks are held protect a visible write. It also adds the contexts reachable from y's context to the escaped set. When the program writes to an instance field (x.f = y), the analysis inspects the state of the object receiving the write. If it is shared and has escaped (implying a visible write), the analysis treats it as if a static field were written into and updates the escaped set. If it is not shared but enters the method through a parameter, the analysis marks the context as *written-to* and places all currently locked contexts in its *locks* set. In all other situations the analysis ignores the write.

At a method invocation, two actions take place. First, if a callee makes a visible write, the analysis marks the caller as *writes*. It then marks *protective* all contexts currently locked as well as the caller's contexts corresponding to the callee's contexts that are locked at the point the visible write occurs. The analysis has marked these contexts in the callee as *protective*. Second, the analysis handles all writes that the callee does not know to be visible writes. If the caller deems a write a visible write, it marks *protective* all currently locked contexts in addition to the caller's contexts matching the callee's contexts that are locked at the point of the write. For easy identification, the analysis has captured these contexts in the callee with the *locks* set. If the caller does not recognize the write as a visible write, it either ignores it or marks the appropriate context *written-to* and adds the locked contexts to its *locks* set.

Five instructions can cause the analysis to add contexts to the escaped set. A store of an object into a static field or a load of an object from a static field implies that the object has escaped the thread, causing the analysis to add all contexts reachable from it to escaped$_{GEN}$. A store of an object into an instance field or a load of an object from an instance field causes all reachable contexts to be escaping if the enclosing object has already escaped the thread. Finally, an object may become escaping as a result of a method invocation. If the corresponding context in the callee is *shared*, the context must escape the thread inside the callee; the analysis adds it to the escaped set.

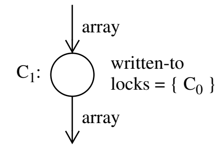
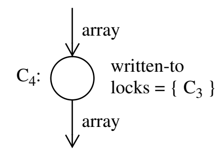
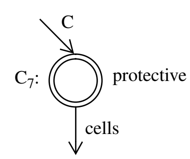
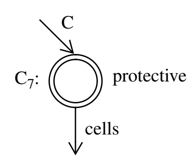
S1: cells[i][j] = value in writeCell	
escaped$_{IN}$ = { reachable(C_0) } guards$_{IN}$ = { C_0 } escaped$_{OUT}$ = { reachable(C_0) } guards$_{OUT}$ = { C_0 }	
S5: C.writeCell(...) in main	
escaped$_{IN}$ = { C_5 } guards$_{IN}$ = { } escaped$_{OUT}$ = { C_5 } guards$_{OUT}$ = { }	no changes to graph
S11: C.writeCell(...) in run	
escaped$_{IN}$ = { reachable(C_2) } guards$_{IN}$ = { } escaped$_{OUT}$ = { reachable(C_2) } guards$_{OUT}$ = { }	
start's implicit call to run	
escaped$_{IN}$ = { reachable(C5) } guards$_{IN}$ = { } escaped$_{OUT}$ = { reachable(C5) } guards$_{OUT}$ = { }	write(start) = T 
S6: start() in main	
escaped$_{IN}$ = { C4 } guards$_{IN}$ = { } escaped$_{OUT}$ = { C4, reachable(C5) } guards$_{OUT}$ = { }	write(main) = T 

Table 6. Data-flow analysis on critical points of the matrix multiplication example.

Only two instructions affect the stack of objects guarding the current critical section. The first, monitorenter x, causes the analysis to push x's context onto the stack since control passes this instruction only when the thread has locked x. A monitorexit instruction unlocks the last locked object, so the analysis pops a context from guards$_{IN}$. This action pairs a monitorexit instruction with the last monitorenter encountered.

4.2 Example

The analysis seeks program points that write to an object or static field while amid a critical section. Our running example has three critical sections: the bodies of readCell, subMatrix, and writeCell. The readCell method does not perform any writes. SubMatrix performs a write, but it writes to sub, a thread-local object. In this case, no other thread will see the result of the write, the escaped set will never contain sub's context, and the analysis will ignore the write (see noteWrite in Table 5). The writeCell method, on the other hand, writes to an object reachable from a formal parameter, causing the analysis to propagate the write up the call chain until it ultimately disqualifies matrix C from being optimized. With the aid of Table 6, we examine this disqualification more closely in the following paragraphs.

Before analyzing writeCell, the analysis places the receiver context (C_0) and all reachable contexts in the escaped set, signifying that these objects may not be thread-local. Moreover, it pushes C_0 onto the guard stack since writeCell is synchronized. Since context C_1, the recipient of the write at statement S1, appears in the escaped set but is not known to be shared, the analysis remembers the write. It marks C_1 *written-to* and stores the currently locked non-thread-local object—C_0—in C_1's *locks* set.

At the point main calls writeCell at line S5, the escaped set only contains context C_5, which corresponds to the args parameter, since the three matrices have not yet escaped the thread. The analysis, having already analyzed writeCell, sees that context C_1 in writeCell is marked *written-to*. Thus, it treats it as if the corresponding context in main (C_8) were written to directly. Since the context C_8 is still local to the thread, the escaped set does not contain it, and the analysis safely ignores the write. As a result, no changes occur to the graph.

When run invokes writeCell on matrix C at statement S11, the analysis again sees C_1's *written-to* flag. Even though the corresponding context in run (C_4) is not shared, it appears in the escaped set. The analysis remembers the write by marking C_4 as *written-to* and stores run's equivalent of C_0 (C_3) in C_4's locks set.

Although not appearing in the Java code, the native start method invokes the run method of class MatrixMultiply. Recall that start's portion of the graph is the same as main's, since the two methods share the contexts marked *shared*. Upon analyzing start, the analysis sees that C_4 receives a write. Since start's corresponding context (C_8) is *shared* and in the escaped set, the write may be visible to other threads. The analysis marks start as a *writes* method and computes all contexts guarding the visible write. In this case, a thread holds C_3's lock when the write occurs, forcing the analysis to mark the caller's equivalent context (C_7) as *protective*. This action signals to the optimizer not to remove the synchronization on this shared context because it protects a visible write.

When main calls start at line S6, the analysis recognizes that the callee performs a visible write. Accordingly, it marks main a *writes* method and notes that all currently locked objects guard a visible write. The write occurs to C_8 while context C_7 is locked, so the analysis marks (again) C_7 as *protective*. Furthermore, since the formal parameters of start are marked *shared*, the objects must escape during the method's lifetime. Hence the analysis adds them to the escaped set.

The analysis continues until it examines all methods and reaches a fixed point in the escaped set and shape graph. In the end, the analysis has marked matrix C's context, but not A's context or B's context, as *protective*, so A and B can be unsynchronized.

5. Transformation

With the aim of speeding up Java programs, the transformation eliminates the unneeded synchronization operations performed on shared objects that do not protect a visible write. We employ the bytecode-to-bytecode transformation described in [7] with the added capability of removing synchronized *blocks*. In short, the transformation clones the defining class(es) of optimizable contexts and makes the cloned class a subclass of the original class. It also clones the paths to the code that constructs these objects as well as to the critical sections they guard.[1] By altering the new and invokespecial instructions, it modifies the allocation sites to construct a new unsynchronized version of the class. Finally, it removes synchronized blocks by turning the monitorenter instruction and all corresponding monitorexit instructions into pop instructions. In general, the transformation proceeds as follows:

1. For each optimizable context, modify the allocation sites, synchronized blocks, and call sites in each method that references the context.
2. For each optimizable context, clone the call chains leading to optimizable allocation sites and critical sections and change those allocation sites and synchronized blocks in the cloned methods.
3. Unsynchronize all classes defining optimizable contexts.

The key to the transformation algorithm is to postpone cloning until all changes have been made to the base code.

Applying the transformation to the matrix multiplication program, we alter the construction of matrix A to read

```
Matrix A = new Matrix$Unsync( nRows, nCols );
```

The new class

```
class Matrix$Unsync extends Matrix {
  ...
  // not synchronized
  public void writeCell( int i, int j, Object value ) {
    cells[i][j] = value;
  }
}
```

is the same as the original class except that its method modifiers lack the **synchronized** keyword. When the program invokes readCell on matrix A, dynamic dispatch finds the unsynchronized version of the method. Furthermore, to unsynchronize the matrix that subMatrix returns, the transformation changes line S8 to

```
Matrix blockOfA = A.subMatrix$Clone1( iOffset, bSize, bSize*k,
    bSize);
```

which invokes the cloned method

```
public Matrix subMatrix$Clone1(...) {
  Matrix sub = new Matrix$Unsync( nRows, nCols );
  ...
}
```

[1] To minimize the amount of cloning, we reuse cloned call chains when possible.

	MatrixMultiply	TreeSearch
Number of classes	497	497
Number of methods	2343	2346
Number of contexts	6326	6256
Number of shared contexts	490	478
Number of instantiated shared contexts	126	126
Number of instantiated shared contexts not protecting a visible write	107	106
Number of optimizable shared contexts	27	28
Number of classes unsynchronized	6	6
Number of methods cloned	2	4
Number of altered allocation sites	29	30
Number of altered call sites	2	4
Number of synchronized blocks removed	0	0
Speed-up	40%	23%

Table 7. Results of the optimization applied to two microbenchmarks.

Just as native methods complicate a program analysis, they pose two problems to cloning during a bytecode-to-bytecode transformation. First, the optimizer cannot modify the body of a native method. Second, a native method's reliance on the interface of a hook method, such as start's dependence on run, rules out cloning because the transformation can alter neither the name nor the signature of the hook method.

This bytecode-to-bytecode transformation may not produce an equivalent optimized program for JVMs or applications that test the anomalies of the current Java Memory Model. A synchronization consists of a lock operation as well as a flushing of a thread's working memory. Eliminating a synchronization operation entirely may no longer ensure visibility across threads. The visible-write analysis identifies when it is safe to remove the locking operation associated with a monitorenter or monitorexit instruction but disregards the flushing. To eliminate only the locking operation, we must be able to split the synchronization operations, a task not expressible at the bytecode level. Performing this optimization in a just-in-time compiler, however, will allow such flexibility. Notwithstanding, most uses of synchronization generally aim to acquire a lock in order to protect the underlying critical section. Using a synchronization operation to refresh unrelated values is generally considered bad practice. Furthermore, current JVMs often ignore visibility issues of the memory model. For these reasons, results from a bytecode-to-bytecode transformation are still valuable at this time, and they indicate the potential of a native code optimization.

Depending on the amount of synchronization and contention in the original program, the optimization may lead to a measurable decrease in overall execution time. Note, however, that the transformation does not remove the locking of unoptimizable objects before they escape the thread. For instance, the call to C.writeCell() at line S5 in the optimized program still acquires C's lock, even though it has not yet escaped the thread. We do not know how much of an impact this has, if any, on typical programs.

6. Evaluation

One would expect that any multi-threaded application that shares data structures in a read-only manner to benefit from this optimization because no lock would be acquired and released for some sections of code. The benefit would be greater if these sections have a large amount of contention in the original program because removal of the synchronization will allow more concurrency in addition to fewer lock accesses.

To look into the effectiveness of the analysis, we have, with the aid of the JavaClass parsing library [15], implemented in pure Java the shape analysis, visible-write analysis, and bytecode-to-bytecode transformation. Using the JDK 1.2.1 with native threads and a just-in-time compiler, we ran the framework on a 400MHz four-processor Sun Ultra Enterprise 450 with 1GB of RAM.

Because of a dearth of freely available full-sized multi-threaded applications, we only report results for two toy problems. The first is the matrix multiplication program found in Figure 1 and referenced throughout the paper, while the second is a multi-threaded binary tree search, written specifically to test this analysis. The latter application constructs a binary tree with 2047 synchronized nodes and starts ten threads searching the tree for a specified match. Each thread examines the left subtree of a node and, if a right subtree exists, spawns a thread to search the right subtree. The threads share the search space but never modify it, making each node of the tree a shared object that does not protect any visible writes.

Table 7 presents some static figures regarding the analysis of these microbenchmarks. The majority of the classes and methods in their transitive closures belong to the JDK core libraries, so the numbers in the table are nearly identical for both programs. The shape analysis creates over 6000 contexts (3 per method), of which approximately 500 represent thread-escaping objects. The analysis identifies allocation sites for 126 of these shared objects. Several reasons account for this seemingly low number. First, some shared contexts represent exceptions that the user program never explicitly throws, so the analysis cannot match them to any allocation sites. Second, many shared contexts originate in native methods, which conceal their allocation sites. Last, since the analysis creates a shared context for all static fields of reference type, some turn out never to be accessed in the program.

Somewhat surprisingly, 107 of the instantiated shared contexts do not protect a visible write. Many are allocated and stored into a static field but never subsequently accessed. These contexts are then optimized if they represent an object that either is defined by a class with at least one synchronized instance method or protects at least one critical section. In MatrixMultiply, the analysis identifies and optimizes matrices A and B. In TreeSearch, the nodes of the search space are optimized. The remaining 26 optimizable contexts primarily encapsulate objects stored into static fields but never used in the program. For example, the class java.util.Collections constructs a random number generator in its static initializer. Even though Random has three synchronized methods, the program never invokes them on this particular random number generator.

The optimized versions of MatrixMultiply and TreeSearch allow more concurrency since the threads do not block to read shared data; however, the reported speed-ups are irrelevant for we can tune the size and thread-count parameters as desired. The important idea is that the analysis recognizes the superfluous synchronization, removes it, and causes a speed-up. A program will in general exhibit a large speed-up if the optimized critical sections are highly contended in the original program.

7. Related Work

It is our understanding that no analyses and optimizations heretofore can eliminate the implicit locks of matrices A and B. Two reasons account for this. First, the shape analyses that seek thread-local objects do not produce an accurate enough picture of shared objects to optimize them. Second, current analyses do not differentiate reads and writes within critical sections.

The work by Bogda and Hölzle [7], Choi *et al.* [8], and Whaley and Rinard [21] seek only thread-local objects, constructing their shape graphs detailed enough to show what objects escape a thread. They essentially view shared objects as a black box. Since the analyses of the second and third papers are flow-sensitive, knowledge of all possible thread interleavings would be necessary to construct an accurate image of shared objects. Without an accurate image, the analyses do not know if two program points refer to distinct shared objects and consequently cannot conclude that a shared object's lock is unneeded.

Ruf [19] has simultaneously and independently produced a shape analysis similar to the one presented here. The main difference is the treatment of program recursion. Consider the invocation $m_2(a_0)$ in method m_1, and suppose m_1 and m_2 belong to the same strongly connected component. In order to eliminate the need for a fixed-point solution, Ruf conservatively unifies the context of a_0 and the context of m_2's formal parameter p_0. Our analysis unifies them only if the context of p_0 flows back into the method m_1. Even though our analysis may identify fewer thread-escaping contexts as a result, the gain on true applications is unclear.

Aldrich and others at the University of Washington [2] present a lock analysis that eliminates redundant locking. At a high level, their analysis propagates the currently locked objects along all paths *down* to monitorenter instructions. This information enables them to remove nested locking of the same lock as well as locking that always occurs while the thread holds another lock. On the other hand, we propagate the objects that are written to and the objects that protect a visible write *up* to callers of the call chain. This information does not capture all ways to reach a critical section. The sets of lock operations removed as a result of these analyses are clearly not equivalent. When a visible write occurs, our optimization keeps all lock operations along all paths to the write even if the program repeatedly acquires the same lock. When a visible write does not occur, our optimization may remove the protecting lock even if an outer lock does not exist. Since these analyses would remove the same lock only by coincidence, they are complementary.

Bogda [6] demonstrates how to use a shape analysis that detects thread-local objects to determine whether or not a method is read-only, where a read-only method is one that never writes to a thread-escaping object. Although his analysis also surveys write instructions, there are major differences between our analyses. First, the analyses capture different information. The read-write analysis seeks only visible writes, whereas the visible-write analysis finds all visible writes inside a critical section that a shared object protects. One can argue that his analysis identifies read-only *methods*, while our analysis identifies read-only *objects*. Second, his analysis is not flow-sensitive, so it does not take into account *when* a shared object escapes the thread. Third, his goal is not to eliminate synchronization but to reduce Java's strong locking to reader/writer locks.

There has been other shape analysis work [9,12,22] outside the area of synchronization elimination. Wilhelm *et al.* [22] present an overview of shape analysis problems and sketches one that asserts properties of the shape graph given its properties at another point in the program. For example, it can show that an acyclic data structure entering a method will still be acyclic upon exiting. The shape analysis presented in this paper would have difficulty proving such situations because it conservatively collapses entire recursive data structures to a single instance. It cannot single out elements of the data structure, but it can recognize different instances of the data structure.

Instead of trying to remove synchronization from Java programs, Flanagan and Freund [11] attempt to detect race conditions in them. In doing so, they need to calculate the set of locks held at each program point, just as we do. To them, however, each lock is given by the expression after the synchronized keyword, and each expression denotes a distinct object. They rely on the programmer to annotate a field with an expression denoting the protecting lock. If this expression does not appear in the lock set at any access to the field, the program has a potential data race. Aliasing causes lock expressions to refer to the same object, thereby causing the analysis to erroneously identify some race conditions. To reduce the number of these false negatives, they make a syntactic substitution for the this keyword. This substitution proves sufficient in the applications they examine, although it is not a general solution. Employing a shape analysis will probably not be able to reduce this number further because the shape graph is a solution to the may-alias problem, not the definitely-alias problem. Flanagan and Freund must be certain an object has been locked at a given program point.

8. Conclusions

To counter the high cost of synchronization in Java, we present an optimization that eliminates a lock when it fails to protect a write to memory accessible to multiple threads. It relies on a flow-sensitive data-flow analysis that determines what writes are visible writes and what objects protect these visible writes. This is possible only with an accurate static picture of shared objects.

Preliminary results reveal that we can remove the locking operations of some globally shared objects. We hope that this analysis—in combination with other analyses, such as one that removes redundant locking—can come closer to eliminating all unnecessary synchronization in Java, leaving only those operations that truly protect writes to shared data under contention.

We are looking into other situations where we can safely remove lock operations. Many programs exhibit phases that alternate reading a shared object and writing to it. Accurate detection of these phases, by recognizing a global barrier, for example, may allow a program analysis to analyze and to optimize each phase in isolation. As a result, more opportunities for concurrency and synchronization elimination may emerge.

Acknowledgments

This work was funded in part by the Stanford/ARPA grant PR-9836 and in part by the NSF grant CCR-9972571.

References

[1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 207-222, Denver, Colorado, 1-5 November 1999.

[2] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Proceedings of the Sixth International Static Analysis Symposium*, Venezia, Italy, September 1999.

[3] David F. Bacon and Peter F.Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 324-341, San Jose, California, 6-10 October 1996.

[4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 258-268, Montreal, Canada, May 1998.

[5] Bruno Blanchet. Escape Analysis for Object-Oriented Languages. Application to Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 20-34, Denver, Colorado, 1-5 November 1999.

[6] Jeff Bogda. Detecting Read-Only Methods in Java. To appear in *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR '00)*, Rochester, New York, 25-27 May 2000.

[7] Jeff Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 35-46, Denver, Colorado, 1-5 November 1999.

[8] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 1-19, Denver, Colorado, 1-5 November 1999.

[9] James C. Corbett. *Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs*. Technical Report ICS-TR-98-20, Department of Information and Computer Science, University of Hawaii, October 14, 1998.

[10] Pedro Diniz and Martin Rinard. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs. In *Journal of Parallel and Distributed Computing* 49(2), March 1998.

[11] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *Proceedings of Programming Languages, Design, and Implementation (PLDI '00)*, pages 219-232, Vancouver, Canada, 18-21 Jun 2000.

[12] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 1-15, St. Petersburg Beach, Florida, 21-24 January 1996.

[13] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley: Berkeley, California, 1996.

[14] Allan Heydon and Marc Najork. Performance Limitations of the Java Core Libraries. In *Proceedings of the 1999 ACM Java Grande Conference*, pages 35-41, June 1999.

[15] JavaClass parsing library version 3.2.3. http://www.inf.fu-berlin.de/~dahm/JavaClass.

[16] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc.: Menlo Park, California, 1994.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley: Berkeley, California, 1997.

[18] Tamiya Onodera and Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 223-237, Denver, Colorado, 1-5 November 1999.

[19] Erik Ruf. Effective Synchronization Removal for Java. In *Proceedings of Programming Languages, Design, and Implementation (PLDI '00)*, pages 208-218, Vancouver, Canada, 18-21 Jun 2000.

[20] Steensgaard, Bjarne. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 32-41, January 1996.

[21] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 187-206, Denver, Colorado, 1-5 November 1999.

[22] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape Analysis. In *Proceedings of Conference on Compiler Construction*, Berlin, Germany, 27 March - 2 April 2000.