

Detecting Read-Only Methods in Java

Jeff Bogda

Department of Computer Science
University of California
Santa Barbara, CA 93106
bogda@cs.ucsb.edu

Abstract. One typically defines a read-only method as a method that does not perform any write operations; however, with respect to a multi-threaded program, we augment this interpretation and let a read-only method be one that does not write to a memory location visible to more than one thread. Such a method can execute concurrently with other read-only methods. With the relaxation of mutual exclusion in mind, we present a read-write analysis that employs the ideas of shape analysis and escape analysis to identify read-only methods in Java. Approximately 31% of the methods in the JDK 1.2 core libraries meets this definition of read-only—nearly 50% more than those that do not perform any write operations.

1 Introduction

Parallel systems of today must be able to execute object-oriented applications efficiently. Programmers write many of these applications in Java [7], a popular object-oriented language that supports multi-threaded behavior at the language level. As a built-in mechanism to protect regions of code from concurrent access, the Java programming language provides the `synchronized` keyword and associates an implicit lock with each run-time object. When applied to an instance method definition, the `synchronized` keyword causes two actions to occur. First, upon entering the method, the executing thread acquires the object's lock and flushes its working memory. Second, upon exiting the method, the thread releases the lock and commits its working memory to main memory. Because of this protecting lock, a thread has exclusive access to any `synchronized` methods invoked on the object.

Depending on the application and the use of a particular run-time object, the above protection may be too strong. It may be meaningful and even beneficial to allow multiple threads to execute some methods concurrently. In particular, code that only reads memory (a reader) can be executed with other code that also only reads memory; however, code that writes to memory (a writer) must have exclusive access to that memory. For example, a multi-threaded program that performs matrix computations can allow threads to read the cells of a shared matrix concurrently.

To implement this multiple-readers/single-writer concept, programmers cannot simply use the implicit locks in Java since they provide a thread exclusive access to an object. Instead, programmers typically resort to run-time libraries that introduce explicit locks in their programs and monitor the invocation of methods. By knowing which methods the run-time system can safely overlap, we can potentially remove this burden from the programmers and still allow the concurrency. For example, at a high

level we can automatically transform a standard program into one utilizing the multiple-readers/single-writer protocol. Alternatively, at a low level the run-time system can reduce Java's implicit locks to read-write locks and allow concurrent readers.

We let a read-only method denote a method that the run-time system can execute at the same time as other read-only methods. Traditionally, because of the ease of detection, this has only included methods that do not perform any write operations. We recognize that some writes are still safe, namely writes to memory locations visible only to a single thread. Thus, in this work a read-only method is a method that does not write to static fields or to fields of objects accessible to more than one thread.

We present a whole-program, static analysis (see Section 3) that determines which methods are read-only. Applying this analysis to the example of Section 2 reveals that 31% of the methods are read-only (see Section 4), a significant improvement over an analysis that uses the traditional notion of read-only.

2 Banking Example

Consider the contrived banking program in Figure 1. After spawning twenty threads that repeatedly get the balance of a shared Account object, the main body repeatedly deposits money into the account and prints the account to standard output. It creates an account at line S2 and instantiates the readers at line S3. By invoking the start method of a ReaderThread object, the program causes the run-time system to spawn a thread that executes the object's run method. Thus at run-time the reading of the account may be arbitrarily interspersed with the deposits. The println routine invokes the Account object's toString method, which the compiler most likely translates to

```
String val = String.valueOf( balance );  
StringBuffer buf = new StringBuffer( val );  
String c = " dollars";  
buf.append( c );  
return buf.toString();
```

The translated version utilizes a StringBuffer object to synthesize the string.

Since the getBalance, deposit, and toString methods of the Account class are declared synchronized, an Account object is thread-safe. As a result, each thread accesses the account atomically, there is a total ordering to the accesses of the account, and each thread is guaranteed to see the most recent amount in the account. Unfortunately this means that a thread cannot read the account while another thread reads it and that a thread cannot necessarily read the balance currently cached in its working memory. Both actions preserve program semantics. The deposit method is the only method that modifies shared memory—memory accessible to more than one thread—and hence needs exclusive access to that memory. The getBalance and toString methods do not make any changes visible to another thread. We propose an analysis that recognizes getBalance and toString as read-only methods.

3 Read-Write Analysis

The analysis presented here identifies write operations to shared memory and labels methods read-only. It traverses, in reverse topological order, the strongly connected components of the static call graph. Upon analyzing each component, it executes three phases: a shape analysis, a thread-escape analysis, and a read-write analysis. Each underlying analysis iterates over the component until it reaches a fixed point. The shape analysis phase estimates the connectivity of heap objects, the thread-escape analysis

```
public class BankExample {
    private static class Account {
        double balance = 0.0;

        public synchronized double getBalance()
            { return balance; }

        public synchronized void deposit( double amt )
            { balance += amt; }

        public synchronized String toString()           // Compiler rewrites using
            { return balance + " dollars"; }           // a StringBuffer object
    }

    private static class ReaderThread extends Thread {
        Account myCopy;

        public ReaderThread( Account acct )
        S1:    { myCopy = acct; }

        public void run() {
            for( int i=0; i<10000; i++ )
                myCopy.getBalance();           // Perform read
            }
        }

        public static void main( String[] args ) {
        S2: Account acct = new Account();

        for( int t=0; t<20; t++ )
        S3:    new ReaderThread( acct ).start();           // Spawn reader

        for( int i=0; i<10000; i++ )
            acct.deposit( 50.0 );           // Perform write

        System.out.println( acct );
        }
    }
}
```

Figure 1. Sample program with a main writer thread and several reader threads.

Table 1. Constraints for the shape analysis phase.

$x = y$ Similarly, $x.f = y$ $y = x.f$	Let the operand \equiv denote the following constraints: $\text{context}(x) = \text{context}(y)$ $\forall f \in \text{fields}(x) \cup \text{fields}(y)$ $\text{context}(x.f) \equiv \text{context}(y.f)$
$\text{foo}(a_0, \dots, a_n)$	$\forall g \in \text{methods-invoked}(\text{foo})$ $\forall i, j, \Phi_i, \Phi_j$, such that $\text{context}(p_i, \Phi_i) = \text{context}(p_j, \Phi_j)$ $\text{context}(a_i, \Phi_i) \equiv \text{context}(a_j, \Phi_j)$

phase subsequently determines which objects may be visible outside the currently executing thread, and the read-write analysis phase determines read-only methods. The phases may be carried out at the same time, although for clarity we discuss each phase as being a distinct step in the analysis. Section 3.2 describes an implementation optimization that combines the shape analysis and thread-escape analysis phases.

We make the following assumptions. We treat an array access as a read or write of a single instance field named `array`, thereby not distinguishing cells of an array. The variables p_0, \dots, p_n represent a method’s formal parameters, while a_0, \dots, a_n denote the corresponding actual arguments at a given call site. Moreover, a method takes its receiver as its first argument (p_0), the destination variable as its second-to-last argument (p_{n-1}), and the thrown exception as its last argument (p_n).¹ Furthermore, we treat a native method specially. If we know its behavior, we hard-code its results into the analysis; otherwise, we assume the worst results.

3.1 Shape Analysis Phase

A shape analysis [5,6,10,13] statically constructs an approximation of the connectivity of heap objects. We employ it at the method level, and, using the terminology presented in the paper on shape analysis by Wilhelm *et al.* [13], we label each method’s resulting snapshot a *shape graph*. A node of a shape graph is an *object context*, which represents one or more objects passing through the method. A directed edge labeled f extends from object context C_i to object context C_j if it is possible that at run-time an object represented by C_j can be accessed through field f of an object represented by C_i .

In presenting the analysis, we follow the format used in [3] and provide a set of constraints (Table 1). Within the constraints, $\text{context}(x.\phi)$ denotes the object context encapsulating any object reachable from the (possibly empty) path ϕ of field dereferences from the variable x . Also, $\text{field}(x)$ is the set of fields of x , and $\text{methods-invoked}(\text{foo})$ is the set of all method implementations named `foo` that may be reached at a given call site. The analysis assumes incoming objects are not aliased, so each formal parameter initially has its own object context. The callers of the method handle any aliasing.

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.

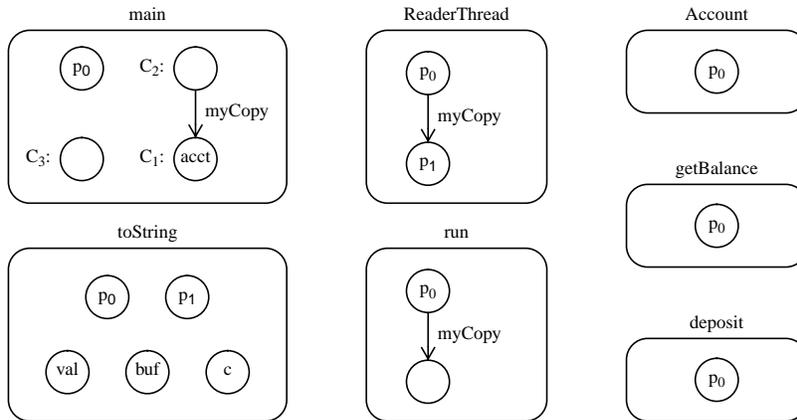


Figure 2. The example's shape graph divided into regions.

The analysis walks the code, manipulating the shape graph as it encounters statements that affect the heap. When it encounters an assignment $x = y$, where x and y are reference variables, the analysis recursively merges the contexts corresponding to x and to y . Across a method invocation, the analysis maps the structure of a callee's formal parameter contexts to the actual parameter contexts in the caller. If a context does not exist in the caller, one is created. To handle program recursion, if the analysis detects that a context C_i may enter the method being analyzed through parameter a_j , it equates C_i and the context of a_j . Since the number of contexts is finite and the actions only merge nodes of the shape graph, the analysis will reach a fixed point.

After the shape analysis inspects our banking example, the shape graph resembles the diagram in Figure 2.² For easy referral, we write the program variable that references an object encapsulated by the context inside the node and an arbitrary label outside. The seven regions of the graph correspond to the seven methods of Figure 1. Since the shape analysis is a backward analysis, each method only knows about objects accessed in the body of the method and objects reachable from formal parameters of called methods.

The final subgraphs corresponding to the `Account` constructor, `getBalance` method, and `deposit` method contain a single context—the context corresponding to the receiver (p_0). The method `toString` uses objects in addition to the receiver, namely a string to hold the balance (`val`), a `StringBuffer` object (`buf`), a string constant (`c`), and the returned object (p_1).

Analysis of the `ReaderThread` constructor reveals that the program stores parameter p_1 into the `myCopy` field of the receiver at line S1. This relationship indicates that the `Account` object is accessible through the `ReaderThread` objects. In `run` the program merely uses the receiver's `myCopy` field.

² For a clearer presentation we ignore exception objects, which do not influence this example, and objects introduced by library classes, which require additional background information.

Table 2. Constraints for the thread-escape analysis phase.

$C.f = y$ (static) $y = C.f$ (static)	Let <i>markShared</i> denote the following constraints: $shared(context(y)) = T$ $\forall f \in fields(y)$ $markShared(y.f)$
$foo(a_0, \dots, a_n)$	$\forall g \in methods-invoked(foo)$ $\forall i, \phi$ such that $shared(context(p_i.\phi))$ $markShared(a_i.\phi)$

In main context C_1 represents the Account object instantiated at line S2, and C_2 encapsulates the twenty thread objects created at S3. The shape analysis maps the structure of the ReaderThread constructor’s formal parameters to the actual parameters, causing C_1 to be attached to C_2 via field myCopy. Context C_3 represents the object denoted by System.out.

3.2 Thread-Escape Analysis Phase

The thread-escape analysis [1,2,3,4,12] phase identifies which objects escape a thread, or, in other words, are potentially shared among threads. Immediately after a thread allocates a new object (*i.e.*, invokes the bytecode new), it holds the only reference to the object. Other threads cannot access this object until the creator thread stores the reference in a portion of the heap accessible to other threads. This occurs with a store into a static field or into an instance field of another escaping object.³ We call such an object *shared* and define the boolean attribute $shared(context(x))$ to be true (T) if an object that variable x references may be shared among threads.

The analysis first assumes that no object escapes (*i.e.*, attribute *shared* is false), and then it traverses the code to find places objects may escape. The constraints given in Table 2 guide the analysis. At a static field access, the analysis marks the context representing the data object—as well as all contexts reachable from this context in the shape graph—as being shared. At a method invocation, the contexts of the objects passed into and returned from a callee inherit the *shared* attributes of the corresponding contexts in the callee. Since the attribute *shared* only changes from false to true, the analysis will reach a fixed point.

Because shared objects are essentially global objects, an invaluable optimization is to reuse the contexts of shared objects. Specifically, if a context C_i in a callee has been marked *shared*, the shape analysis can recursively merge the caller’s corresponding context with C_i instead of duplicating C_i ’s structure in the caller. This requires us to combine the shape analysis and thread-escape analysis phases but greatly reduces the space and time usage of the overall analysis.

³ A store into a static field means that multiple threads *can* access the object but does not necessarily imply that multiple threads *will* access the object. A more precise, expensive analysis may be able to ignore the innocuous static field accesses.

Table 3. Constraints for the read-write analysis phase.

x.f = y (instance)	Let <i>markWrittenTo</i> denote the following constraints: if shared(context(x)) write(m) = T else written-to(context(x)) = T
C.f = y (static)	write(m) = T
foo(a ₀ , ..., a _n)	$\forall g \in \text{methods-invoked}(\text{foo})$ if write(g) write(m) = T $\forall i, \varphi$ such that written-to(context(p _i , φ)) markWrittenTo(context(a _i , φ))

Once this phase has reached a fixed point, it has identified those objects, from the perspective of a given method, that may be accessed by more than one thread. When applied to the running example, the analysis deems the contexts labeled C_1 , C_2 , and C_3 (Figure 2) as shared. C_3 is shared because it encapsulates the object whose reference is stored in the static field `System.out`. Context C_2 is shared because the program invokes the native method `start` on it, which we know makes the thread object accessible to multiple threads. Since context C_1 is reachable via field `myCopy` from context C_2 , it also is shared. This signals that the `Account` object, the `ReaderThread` objects, and standard output are shared objects and that the memory these objects enclose is global memory. The thread-escape analysis does not mark any other contexts *shared*.

Since the `String` and `StringBuffer` objects of the `toString` method are not shared and not reachable from a formal parameter, they are *thread-local* objects. The memory they enclose are accessible only to the thread executing `toString`. The read-write analysis phase of the next section allows read-only methods to write into fields of these thread-local objects.

3.3 Read-Write Analysis Phase

The last phase, the read-write analysis phase, determines if a method writes to the heap in such a way that more than one thread can see the effects. We term such a write a *visible write*. Two situations characterize a visible write: a write to a static field and a write to an instance field of a shared object. A method in which no visible write occurs during its execution is a *read-only method*; otherwise, it is a *write method*. This phase initially assumes all methods are read-only and seeks places where visible writes may occur.

The constraints in Table 3 guide the analysis of a method m . We let the boolean attribute *write* be true for m if it or one of its callees writes to a shared object and the boolean attribute *written-to* be true for an object context if the program writes to one of its fields. Initially these attributes are false. When the analysis encounters a write to a

static field, it recognizes that the result of the write is visible to other threads and marks m a write method. When the analysis encounters a write to an instance field, it inspects the context representing the base object. If the thread-escape analysis marked its context *shared*, signifying that a portion of the heap is potentially accessible to other threads, m is a write method; otherwise, the analysis sets its context's *written-to* attribute to true. When the analysis discovers a method invocation, it deems m a write method if any callee is a write method; otherwise, it treats the *written-to* contexts in the callee as if the corresponding contexts in m were directly written to. If any of these contexts are marked *shared*, m is a write method. Since the *written-to* and *write* attributes only change from false to true, the analysis reaches a fixed point.

The read-only nature of a method depends on its calling context. A method is always read-only if its *write* attribute is false and if no object context reachable from the contexts of the formal parameters (excluding p_{n-1} and p_n) has its *written-to* attribute set to true. Under no circumstance will such a method perform a visible write. If a method's *write* attribute is false, but it writes to an incoming object, it is read-only if the incoming object is a thread-local object. Therefore, an accurate response to the question "Is this method read-only?" requires us to conservatively assume that all incoming objects are shared.

Consider the results of the read-write analysis on our example. The method `getBalance` is trivially a read-only method because no writes occur. The method `toString` only writes to thread-local objects—the `String` and `StringBuffer` objects—and hence is also always read-only. On the other hand, method `deposit` writes to a field of an incoming parameter (the receiver), making it a read-only method only if the receiver is thread-local. Since `main` invokes `deposit` on a shared object, the `main` method is a write method. It would be safe (although less precise) to mistake `getBalance` and `toString` as write methods, but it would be incorrect to deem `main` a read-only method.

4 Evaluation

We ran the analysis on the transitive closure of the banking example using the JDK 1.2.1. Even though these results apply only to this example, we feel they are indicative of results of the core classes pulled in by any multi-threaded application. Table 4 presents the number of read-only methods that the read-write analysis identifies and, for the curious reader, compares these numbers with the number of synchronized methods.

The first column of the table lists the ten classes (with more than ten methods) that have the highest percentage of read-only methods. The second column gives the total number of analyzed non-native methods in the class listed in column one, while column three shows the number of these that are declared synchronized. In the fourth and fifth columns we show the results of an analysis that does not allow a read-only method to perform any writes into the heap. This corresponds to the traditional notion of read-only. The fourth column gives the total number of read-only methods for each class, while the next column shows the number of these read-only methods that are synchronized. The columns entitled "Writes to Local Objects" present the same figures but for the case where read-only methods are allowed to write to thread-local objects. Here all incoming arguments are conservatively assumed to be shared. The final two columns

Table 4. Top JDK library classes, in terms of the percentage of read-only methods.

Class	Methods Analyzed	Methods Sync.	Read-Only Methods					
			No Writes		Writes to Local Objects		Writes to Parameters and Local Objects	
			Total	Sync.	Total	Sync.	Total	Sync.
Dec.FormatSymbols	20	0	16	0	17	0	17	0
Character	12	0	10	0	10	0	10	0
GregorianCalendar	30	0	18	0	19	0	23	0
Double	13	0	8	0	8	0	9	0
Signature	12	0	3	0	7	0	10	0
SimpleTimeZone	16	1	6	1	8	1	9	1
BigInteger	25	0	6	0	12	0	22	0
BitArray	15	0	5	0	7	0	10	0
Inflater	13	10	4	4	6	6	7	7
ClassLoader	11	0	3	0	5	0	5	0
Total of all classes	2217	105	481	12	694	18	1024	38
Total without constr. and static initializers	1743	105	442	12	567	18	773	38

show results for the case where all incoming arguments are thread-local objects. The last two rows give the totals for the entire set of 498 classes. The last row ignores constructors and static initializers.

In all, 694, or 31%, of the 2217 methods are always read-only, and 46% are read-only if we know that the objects entering the methods are not shared. In the worst case our analysis recognizes 44% more read-only methods than a traditional analysis, which identifies 481 read-only methods. Constructors, which typically do not require synchronization, account for 17% of the write methods, and static initializers, which are automatically synchronized by the run-time system, account for 6%.

The low percentage of synchronized methods in the core libraries suggests that the classes either entrust mutual exclusion to the programmer or realize mutual exclusion through synchronized blocks. In the latter case, our analysis would greatly profit from the examination of synchronized blocks in addition to synchronized methods. Contrary to this observation, the class `java.util.zip.Inflater` contains mostly synchronized methods. In fact, all six of the methods deemed read-only are synchronized, suggesting that the locking mechanism is stronger than necessary for these methods.

An optimization can use these results to know when it is safe to overlap execution. Suppose we have a transformation program that automatically replaces Java's synchronization mechanism for the class `Account` with high-level, reentrant read and write locks that implement the multiple-readers/single-writer paradigm. These locks could be built on top of Java's synchronization primitives to ensure visibility across threads. The read lock would monitor the use of the `getBalance` and `toString` methods. For example, the code

```
public double getBalance() {
    readLock.acquire();
    try { return balance; }
    finally { readLock.release(); }
}
```

may reflect an optimized `getBalance` method. Similarly, the write lock would provide mutual exclusion for the `deposit` method. For this example, optimizing the `Account` methods will increase concurrency enough to de-emphasize the overhead of managing high-level locks. In general, however, it is difficult to know which classes will benefit from this optimization.

5 Related Work

As far as we know, our read-write analysis is the first of its kind to be applied to Java. Researchers have thoroughly studied memory access patterns for programs written in C and Fortran, especially in the context of parallelizing compilers. It is our understanding that these studies have focused mostly on inner loops and array accesses of non-object-oriented languages.

Much of the research requires programmers to annotate their code to specify the access patterns of objects. For example, through the keyword `const` [11], C++ programmers identify objects that are not written to and member functions that do not modify the state of their receivers. To verify that a program respects its `const` declarations, the compiler must inspect the writes and invoked methods just as our analysis must.

The work of Yasugi *et al.* [14], which directly applies to a variant of Java, treats a read-only method as a method that does not write to the instance fields of its receiver. To eliminate the difficulty of aliasing, they rely on the auxiliary keyword `internal`. Once they determine the status of a method, they decrease the size of critical sections and allow concurrent reading of the receiver's state. Also, they remove mutual exclusion altogether if they conclude a method reads unchangeable fields of the receiver.

With VJava [8], an extension to the Java programming language, the run-time system wishes to allow multiple readers of an object. VJava achieves this by requiring the programmer to specify the view a thread will have of a shared object. The view indicates the fields that the thread will read and write. If two views do not write to the same part of an object, the run-time system decides that the threads can access the object concurrently.

An analysis that does not require annotations is the commutativity analysis of Rinard and Diniz [9]. Their analysis groups operations and determines which groups can be ordered arbitrarily. They have applied their analysis to a subset of C++ in order to automatically parallelize code.

6 Conclusions

Parallel systems look to speed up synchronization and to maximize concurrency. Recent Java research has provided static analyses that determine which objects will *not*

be shared by multiple threads. With knowledge of such objects, an optimizer may remove lock accesses on these objects. We turn our attention to objects *shared* among threads and strive to speed up multi-threaded programs by allowing threads to execute methods concurrently.

We defined a read-only method to be one in which it and any called methods do not write to a static field or into a shared object. We then described a whole-program static analysis that identifies read-only methods. Results show that approximately 31% of the non-native methods in the JDK 1.2 core libraries meets our definition of read-only. One potential improvement is the ability to analyze synchronized blocks in addition to synchronized methods. As multi-threaded programming in Java gains popularity, more thread-safe libraries will emerge, and the number of synchronized read-only methods will increase.

Identification of read-only methods should give rise to optimizations that allow efficient concurrent access to objects. We know that overlapping the execution of read-only methods can speed up a program, but the question of which methods and which shared objects will benefit most from this optimization still remains.

Acknowledgments

We thank the anonymous reviewers of LCR for their valuable comments on earlier versions of this paper. This work was funded in part by the Stanford/ARPA grant PR-9836 and in part by the NSF grant CCR-9972571.

References

1. 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 (SAS '99)*, Venezia, Italy, September 1999.
2. 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.
3. 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.
4. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. 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.
5. 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, 14 October 1998.
6. 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.

7. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley: Berkeley, California, 1996.
8. Ilya Lipkind, Igor Pechtchanski, and Vijay Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 447-460, Denver, Colorado, 1-5 November 1999.
9. Martin C. Rinard and Pedro C. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems*, pages 942-991, Volume 19, Number 6, November 1997.
10. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 16-31, St. Petersburg Beach, Florida, 21-24 January 1996.
11. Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley: Berkeley, California, 1997.
12. 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.
13. Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape Analysis. In *Proceedings of the Conference on Compiler Construction*, Berlin, Germany, 27 March - 2 April 2000.
14. Masahiro Yasugi, Shigeyuki Eguchi, and Kazuo Taki. Eliminating Bottlenecks on Parallel Systems using Adaptive Objects. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 80-87, Paris, France, 12-18 October 1998.