Proceedings of the Java™ Virtual Machine Research and Technology Symposium (JVM '01)

Monterey, California, USA
April 23–24, 2001

Can a Shape Analysis Work at Run-time?

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

Abstract

A shape analysis is a whole-program analysis that can identify run-time objects that do not need to be placed in the global heap and do not require any locking. Previous research has shown that these two optimizations can speed up some applications significantly. Unfortunately, since a shape analysis—like any whole-program analysis—requires *a priori* knowledge of the complete call graph, it has not been implemented in a JVM, which essentially builds the call graph as a program executes. In this paper, we adapt an efficient shape analysis to be incremental so that it can analyze an executing program. We investigate trade-offs regarding three approaches to performing the analysis inside a JVM and report results on a number of applications. Our measurements suggest that such an analysis may be viable if it uses results of previous executions and if it delays the initial analysis until the end of the first execution.

1. Introduction

A shape analysis [17] is a static whole-program analysis that conservatively predicts the connectivity of heap objects. It proves useful for escape analyses because it identifies objects that may "escape" a method and that may be accessible to more than one thread. Such knowledge steers optimizations such as synchronization elimination and stack allocation.

Unfortunately, shape analysis, as described heretofore, ignores the true flavor of Java programs. The Java programming language prides itself on its dynamic loading and binding, yet a shape analysis requires *a priori* knowledge of all classes. This requirement stems from the fact that it must know all potential targets of a call site. Without the knowledge of potential targets, it is forced to be overly conservative. The many programs that use dynamic loading or that rely on dynamic program attributes—such as the classpath variable—will not benefit from optimizations dependent on the results of a whole-program shape analysis.

Even without Java's dynamic features, an optimizer faces a difficulty. It cannot express, in bytecode, the removal of lock operations or the stack allocation of objects—exactly those optimizations that the shape analysis enables. Researchers sidestep this problem either by annotating the bytecode with suggestions to the JVM or by translating the program into a language in which these optimizations are expressible.

The only true solution to these limitations is to perform the shape analysis at run-time. By operating while the analyzed program executes, the analysis can *observe* (as opposed to calculate) the classes the JVM dynamically loads. More importantly, it can observe the targets of a call site, yielding more precise results than an off-line analysis would. Last, it is not restricted to optimizations expressible in bytecode and can perform optimizations in a JVM-dependent manner.

Despite these advantages, a dynamic shape analysis faces several difficulties. It incurs a run-time cost and must work with incomplete information. Since it is by nature a whole-program analysis, it must build upon and modify previous results as new information arrives. Consequently, a previously optimized object may no longer be optimizable, requiring the optimizer to undo optimizations before an erroneous execution results.

To address these difficulties and trade-offs, we present and evaluate three ways to perform a shape analysis at run-time. The first approach begins the interprocedural analysis as soon as a program starts to run. The second delays the interprocedural analysis until the run-time system has seen a portion of a program's execution. Finally, the third approach reuses analysis results from previous executions. We discuss the advantages and disadvantages of each strategy.

In short, this paper offers:

- An incremental version of an efficient shape analysis;
- Experimental results illustrating the inherent difficulties of employing a shape analysis dynamically;
- A comparison of three approaches to performing the incremental analysis; and
- Insight into making the analysis viable.

```
for each strongly connected component (SCC) in rev. top. order
    for each method m in the SCC
        analyze m intraprocedurally
        for each call site s in m
            for each target t of s
                if t and m are in the same SCC
                    unify actuals of s and formals of t
                else
                    propagate from t to s
```

Figure 1. Static shape analysis algorithm.

After outlining an efficient whole-program shape analysis (Section 2), we adapt it to be incremental (Section 3). In doing so, we recognize that the analysis can classify objects with varying degrees of locality. In some cases, it can guarantee that an object will be local to a thread regardless of future program paths and class loadings. In Section 4 we show the results of an empirical study that compares the aforementioned ways to perform the analysis. Last, Section 5 presents related work, and Section 6 presents conclusions, two related open problems, and future work.

2. Whole-Program Shape Analysis

This section sketches a conservative, but efficient, version of a whole-program shape analysis based on the analyses described in [4] and [10]. Section 3 adapts the algorithm presented here to be incremental. Since the emphasis of this paper is on performing a shape analysis at run-time, not on the efficacy of the analysis itself, we omit strategies one can use to improve precision. For a complete description of the problem, we refer the reader to [1,3,4,5,6,10,16].

A shape analysis is an interprocedural data-flow analysis that approximates the run-time structure of heap objects and identifies objects potentially reachable from a static field. Figure 1 presents the algorithm at a high level. The results of this analysis dictate when it is safe to perform certain optimizations.

We view the results of a shape analysis as a graph. A node in the graph is an abstraction of one or more run-time objects. An edge in the graph represents an instance field dereference and is labeled with the name of the field. The analysis associates each program variable with a node in the graph and connects nodes to reflect the structure of the heap. In the end, if the analysis has associated two variables within a method with distinct nodes, it guarantees that these variables can never reference the same object at run-time. Furthermore, if the analysis has associated a variable with a node that is marked *shared*, the analysis believes the variable may reference an object reachable from a static field. It guarantees that a node not marked *shared* represents thread-local objects. Such objects cannot be accessed by multiple threads and are subject to thread-local optimizations.

```
import java.util.*;

public class Example
{
  public static void main( String[] args )
   throws ClassNotFoundException, InstantiationException,
     IllegalAccessException
  {
    String className = args[0];
    Class theClass = Class.forName( className );
    List listImpl = (List)theClass.newInstance();
   String type = args[1];
   test( listImpl, type );
  }

  private static void test( List list, String element )
  {
    if( element.equals( "int" ))
      for( int i=0; i<10; i++ )
        list.add( new Integer( i ));
    else
      for( int i=0; i<10; i++ )
        list.add( null );
  }
}
```

Figure 2. Example program with explicit dynamic loading.

Consider the example in Figure 2. As input to the small program, the user specifies a class that implements the List interface as well as the type of an element (either integer or null). The program instantiates the specified class and repeatedly inserts elements of the specified type into the container. One may write such a program in order to compare the efficiency of various List data structures.

An intraprocedural phase analyzes each method by performing a data-flow analysis on the stack-based bytecode, unifying corresponding nodes at control-flow merges. The intraprocedural analysis of test reveals a very simple picture of the heap (see Table 1). All of test's variables refer to distinct nodes, and the method does not reveal the structure of these nodes. The nodes labeled list and element statically encapsulate the two formal parameters, the node labeled exception corresponds to the exception object that the method may throw, the "int" node denotes the global String constant "int", and the node labeled integer corresponds to the Integer objects appended to the list. A thick border

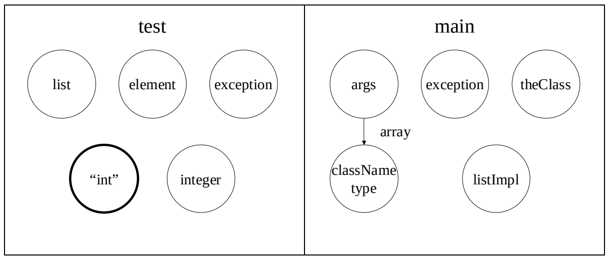

Table 1. Portions of the graph corresponding to the methods test and main.

signals that the "int" node is *shared* and hence may be accessible to multiple threads. Similarly, the intraprocedural analysis of main (also Table 1) reveals five nodes—one for the exception object, one for the listImpl object, one for the Class object, one for the incoming array, and one for the contents of the incoming array. Since a static analysis generally cannot distinguish elements of an array, the variables className and type map to the same node. These subgraphs serve as summaries for the interprocedural portion of the analysis.

The context-sensitive interprocedural phase connects call sites to target methods by mapping the structure of the formal parameters to the corresponding actual parameters. If the caller and callee are in the same strongly connected component (SCC)[1] of the call graph, the analysis merges the node of an actual parameter with the node of the corresponding formal parameter. This unification obviates the need to iterate over the SCC until a fixed point is reached, but it introduces some conservatism. If the caller and callee are in separate SCCs, the analysis imposes the structure of the formal parameters on the actual parameters. However, if the analysis has marked a node in the callee as *shared*, it merges the node with the corresponding node in the caller. This ensures that all methods work with the same *shared* nodes.

The shape analysis is a backward analysis in that it examines a target method before examining a caller method. To accomplish this, it constructs a static call graph and examines each SCC in reverse topological order. Within an SCC, it examines methods arbitrarily.

Without additional information, it is impossible to construct a complete static call graph for our example since the class implementing the List interface is not known statically. This is due to the forName method, which dynamically loads the type of the list, and to the classpath variable, which gets defined only at run-time. Figure 3 shows an incomplete call graph for our example. The targets of the add invocations cannot be deduced from the program's text. In this case the analysis can give up, can conservatively assume that all objects passed to add become *shared*, or can somehow guess the target methods. The last option may lead to incorrect results.

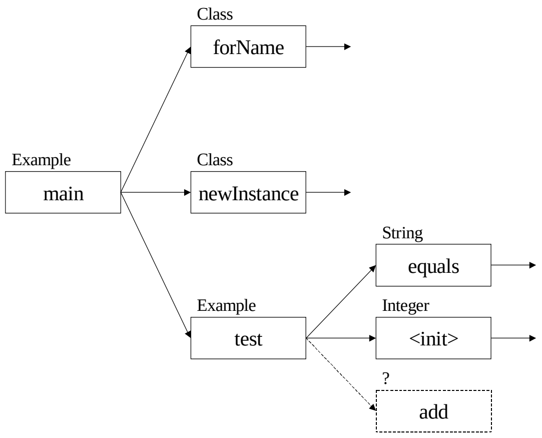

Figure 3. Incomplete call graph of our example.

By performing the analysis at run-time, we can solve the problem of not knowing the call graph because a dynamic analysis can observe the target of the add method. The next section presents an adapted version of the analysis, which can operate dynamically in a JVM.

3. Incremental Shape Analysis

We adapt the above shape analysis to work while the program executes. To be effective, it must work with an incomplete call graph and, as the call graph expands, build upon and modify previous results.

A dynamic analysis avoids the problems caused by dynamic loading and binding because it can observe the targets of call sites. At the same time, this enables it to be more precise than a static analysis for two reasons. First, it knows the exact target(s) of a call site, whereas a static analysis generally amasses a conservative set of potential targets.[2] Second, it only propagates information to call sites that the program executes. For instance, in our example, the dynamic analysis does not need to analyze both calls to add. The next section will explain why this is the case.

[1] A *strongly connected component* is a maximal set of nodes in which there is a path from any node in the set to every other node in the set.

[2] The degree of conservatism depends on the method resolution scheme that the static analysis employs.[14]

```
AnalyzeInvocation( CallSite s, Method t )—
  // step 1
  if <s,t> already analyzed, return
  if t has not been analyzed, analyze t intraprocedurally

  // step 2
  if caller and t are currently in the same SCC
     unify actuals of s and formals of t
  else
     record <s, t>
     DetermineChangesToSCCs( s, t )

  // step 3
  if caller and t are in the same SCC
     PropagateChangesUpCallGraph( t )
  else
     PropagateChangesAlongBinding( s, t )

DetermineChangesToSCCs( CallSite s, Method t )—
  // collapse cycles in call graph involving binding <s,t>
  if RecursivePathExists( method containing s, t, set )
     unify actuals of s and formals of t
     add SCC of t to set

  union all SCCs in set

RecursivePathExists( Method m, Method t, Set set )—
  // recursively collapse cycles in call graph
  if m equals t return true

  sameSCC = false
  for each binding <s,g> such that g and m are in the same SCC
     if RecursivePathExists( method containing s, t, set )
        unify actuals of s and formals of g
        add SCC of m to set
        sameSCC = true

  return sameSCC

PropagateChangesUpCallGraph( Method t )—
  // push changes to all call sites targeting t's SCC
  for each binding <s,g> such that g and t are in the same SCC
     propagate from g to s
     if change occurred to formals of method containing s
        add method containing s to set

  for each m in set
     propagateChangesUpCallGraph( m )

PropagateChangesAlongBinding( CallSite s, Method t )—
  // push summary information from t to s
  propagate from t to s
  if change occurred to formals of method containing s
     propagateChangesUpCallGraph( method containing s )
```

Figure 4. Incremental shape analysis algorithm.

3.1 General Approach

The general algorithm for the incremental analysis appears in Figure 4. Because it is incremental, it does not know the entire call graph at the time of analysis. It consequently works with what it does know and modifies the results as the call graph grows.

The binding of a method to a call site drives the analysis. When a new binding occurs, it performs three steps (see AnalyzeInvocation in Figure 4). First, if it has not already analyzed the target method intraprocedurally, it does so at this time. Second, it identifies any changes to the SCCs as a result of this binding. Third, it propagates the summary of the target method up the call graph. We describe each of these steps in more detail below.

Just as it does in the whole-program version, the analysis first analyzes a method intraprocedurally. The structure of the nodes of the formal parameters will serve as a method summary when the analysis propagates information across call sites. In general, we must maintain information regarding nodes not reachable from the nodes of formal parameters since they may become reachable at a later time. The cost of analyzing a method is a one-time cost; the analysis never needs to analyze it again. Therefore, this cost resembles the cost of bytecode verification and will be nearly negligible for most applications.

To conveniently find the callers of a target method that are outside the target's SCC, the analysis maintains an abbreviated call graph, in which edges between methods within the same SCC are omitted. As the analysis adds edges to this call graph, the SCCs may change. Starting at the new target method, the analysis does a reverse depth-first traversal of the abbreviated call graph. If it

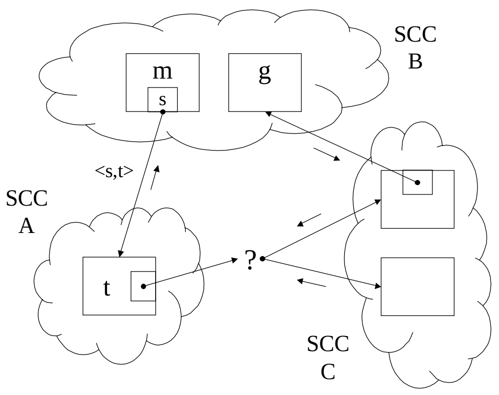

Figure 5. The detection of cycles in the call graph involving the new binding <s,t> (DetermineChangesToSCCs).

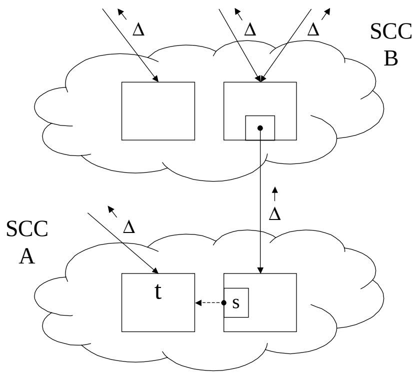

Figure 6. The recursive propagation of summary information along all edges entering t's SCC (PropagateChangesUpCallGraph).

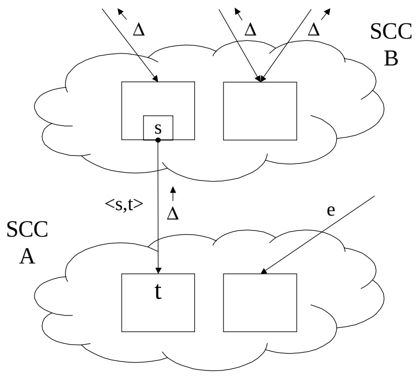

Figure 7. The recursive propagation of summary information starting with the binding <s,t> (PropagateChangesAlongBinding).

returns to the target method, indicating that recursion may occur, it unions all SCCs of the methods along this path and collapses the involved call sites. Figure 5 attempts to illustrate this step. Beginning with the edge <s,t>, the short arrows indicate the traversal of edges between SCCs. Depending on the connectivity of the abbreviated call graph, this may be an expensive operation. In the worst case, the analysis inspects every edge.

The third step propagates the summary of the target method up the call graph. If the caller and the callee are in the same SCC, the analysis must propagate along all call sites that invoke any method in the target's SCC; otherwise, it only needs to propagate along the new binding. Figure 6 depicts the former case, and Figure 7 the latter. Note that the analysis does not propagate along edge e since no changes can occur within t's SCC as a result of the new binding. In either case, the caller then acts as the callee, and its summary flows to its callers. This process continues until either the analysis reaches a root method of the call graph or the structure of the formal parameters of the method containing the call site does not change. In the worst case, the number of propagations is equal to the number of bindings, although we have observed that the test for a change in the formals greatly reduces the number of propagations. Nonetheless, the number of propagations can be several times the number of bindings, as we will see in the next section.

Consider our example in Figure 2 and suppose the program instantiates the class java.util.Vector and adds null references to the list. Figure 8 shows the order in which the incremental analysis constructs and examines the call graph. For clarity, we ignore methods not shown in the diagram. Each step is numbered, where a number inside a box denotes an intraprocedural analysis and a number outside a box indicates a propagation to a call site.

The analysis starts by analyzing main. When the program invokes the forName method, the analysis examines forName in class Class and propagates the information to the call site in main. Next, when the program invokes newInstance, the analysis examines the method and propagates its summary to main. The method newInstance, in turn, calls the constructor of class Vector, <init>. The analysis intraprocedurally analyzes <init>, propagates the summary of <init> to newInstance, and propagates the changes in newInstance to main. The process continues until no changes to the call graph occur.

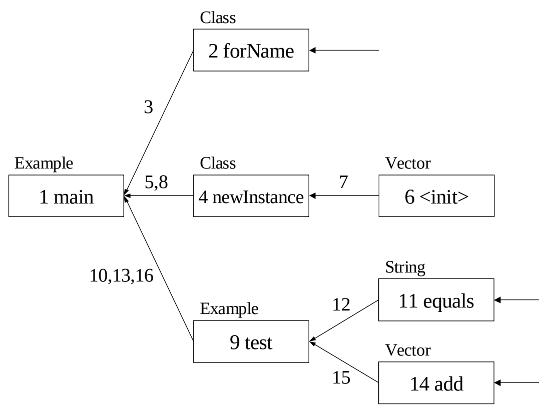

Figure 8. Steps taken by the incremental analysis on our example.

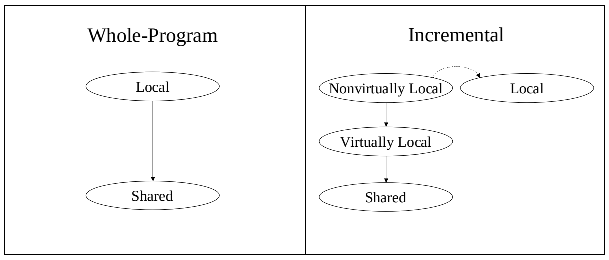

Table 2. Lattices describing the hierarchy of object classifications for the whole-program and incremental approaches.

Since the program never takes the first path of the if-test, the analysis never analyzes the Integer constructor and never propagates information to the first call to add. An off-line whole-program analysis would conservatively do both because it cannot predict the run-time flow of control.

3.2 Classification of Objects

The whole-program version characterizes a node as either *local* or *shared*. A node is initially *local* and becomes *shared* if the analysis merges it with a *shared* node or if it becomes reachable from a *shared* node (see the first column of Table 2). The incremental version introduces two more classifications in order to determine if an object is *guaranteed* to be thread-local, regardless of future classes loaded into the system. We describe all classifications below.

Local—A node is *local* if, no matter what path the program takes and what new classes the system loads, it can never be reachable from a *shared* node.

Nonvirtually Local—A node is *nonvirtually local* if it is not reachable from a node of a variable that is passed into or returned from a virtual method (*i.e.*, it flows into no virtual methods). This node is currently thread-local but may become shared as new methods execute.

Virtually Local—A node is *virtually local* if it flows into a virtual method. In general, a *virtually local* node is not guaranteed to be thread-local because the target of a virtual call site may change in future executions. Therefore, the analysis never knows if it has seen the entire scope of the node.

Shared—A node is *shared* if it is reachable from a *shared* node. The analysis initially marks a node associated with a static field as *shared*.

The relationship of these classifications can be viewed as the second lattice in Table 2. Initially a node is *nonvirtually local*. If it flows into a virtual method, it becomes *virtually local*. If it is stored into or read from a static field, it becomes *shared*. The unification of two nodes takes the meet of the classifications, as defined by the lattice. Additionally, the analysis upholds the rule that a node is at least the lowest type of any of its parents in the resulting graph. Therefore, any node reachable from a *shared* node must also be *shared*, and no node reachable from a *virtually local* node may be *nonvirtually local*.

The classification of *local* is a special case. A node changes from *nonvirtually local* to *local* if, for every method to which the node flows, the method's corresponding node is *local*. This guarantees that *local* nodes can be optimized without later being deoptimized. In our implementation, each *nonvirtually local* node maintains a list of nonvirtual methods to which it flows. After analyzing the binding to one of these methods, we remove the method from the list if the corresponding node in the target is marked *local*. When no methods remain in the list, we promote the node to *local*.

In our example, the node associated with element is *virtually local* because it flows (as the receiver) into the virtual method equals. Similarly, the node associated with variable list is *virtually local* because it flows into the virtual method add. During the initial propagation from test to main, the analysis marks the node associated with listImpl as *virtually local*.

4. Evaluation

An on-the-fly analysis will only be worthwhile if the speed-up resulting from optimizations offsets the time required to carry out the analysis. We assume that an optimization—such as synchronization elimination or stack allocation—depends on the identification of a thread-local object. Thus, in general, the more thread-local objects the analysis identifies, the better.

The optimizer's strategy greatly influences the number of these objects. We label an optimization *pessimistic* if it optimizes an object only when the analysis guarantees that multiple threads cannot access the object. This corresponds to optimizing only objects represented by nodes marked *local*.

In contrast, we label an optimization *optimistic* if it assumes that an object is thread-local before the analysis has analyzed the entire scope of the object. This corresponds to optimizing objects represented by nodes marked *local*, *nonvirtually local*, or *virtually local*.

This distinction affects not only the number of optimizations but also the number of deoptimizations. Consider the following code:

```
x = new
lock x
foo( x )
unlock x


static void foo( p )
{
   X.global = p;
}
```

Suppose foo has not been analyzed at the time of the allocation of x. An optimistic approach immediately optimizes the new object and probably removes the subsequent lock operation. However, after analyzing foo, which makes the new object escape the thread, the optimizer must undo the optimization of x before the body of foo executes. On the other hand, a pessimistic approach waits until it sees the entire scope of x. In this case, it does not optimize x because it has not seen the method foo at the time of the allocation. If foo were not to make x visible to other threads, the pessimistic approach would miss out on the optimization.

The number of optimizable objects also depends on when the dynamic analysis begins. If it starts when the program starts, the optimizer has the potential to capture all optimizable objects. On the other hand, if it starts in the middle of program execution, the optimizer may miss some optimizations. The overall cost of the analysis is smaller the later the analysis begins, thereby encouraging the run-time system to postpone commencement.

We investigate this trade-off by evaluating three approaches. The first begins the analysis immediately in order to capture all optimizable objects. The second delays the analysis a predetermined amount of time, in the hope of reducing the overhead. Finally, the third approach reuses the propagation results of previous analyses to counter the high run-time cost without sacrificing optimizable objects. Before we elaborate on these approaches, we describe our experimental framework and benchmark applications.

4.1 Experimental Framework

We implemented an incremental shape analysis in Java. To allow the analysis to operate on an executing application, we instrument the application to invoke the analysis before key points in its execution. These points are method entry, method exit, call sites, allocation sites, and monitorenter instructions.[1] This instrumentation enables us to identify previously unseen call site/target pairs and to monitor object allocation. We do not trace the execution of system threads; hence all measurements ignore start-up. We ran all tests on a 400MHz Pentium II, using Sun's Java 2 (build 1.3.0-C) for Windows.

Since the JVM disallows both instrumented and uninstrumented versions of core JDK classes, the analysis code is also instrumented. This makes it impossible to determine the running time of our implementation. Notwithstanding, by counting the propagations, we can still sense the overhead of the analysis.

4.2 Benchmarks

Our benchmark suite consists of *jess*, *db*, and *mtrt* from the SPECjvm98 benchmarks [12]; *JLex*; *java_cup*; *slice*; and *volano*. For completeness we include our example program in the first two tables. The multi-threaded *slice* applet, obtained from [10], visualizes radiology data. *Volano*, a multi-threaded chat room simulation, is the client side of VolanoMark 2.1.2.[15]

Table 3 lists the executions we used in our experiment. Some results varied slightly from one run to another, depending on the behavior of the threads. We ran *JLex* on *sample.lex*, which was included in its distribution, and on a homework solution for a compilers class. Similarly, we ran *java_cup* on Java 1.1's grammar, which came with its distribution, and on a homework solution.[2] Due to the significant slowdown caused by our instrumentation, we ran the SPECjvm98 applications with the smallest size input (-s1).

The third column in the table is the number of method invocations during execution of the instrumented program and is indicative of the running time. The number of distinct methods executed, the fourth column, ranges from 18 for our small example to over 1400 for *slice*. This number equals the number of intraprocedural analyses needed. The column "# SCCs" lists the counts of strongly connected components, which are close to the figures of the previous column. This means that few call chains form recursive paths and suggests that the unification of call sites may not lose much, if any, precision. The second-to-last column

[1] The bytecodes invokevirtual, invokespecial, invokeinterface, and invokestatic call methods, and new, newarray, anewarray, and multinewarray allocate heap objects.

[2] The homework input files are available on the first author's homepage (http://www.cs.ucsb.edu/~bogda).

Benchmark	Description	# Method Invocations	# Methods	# SCCs	# Call Site/ Target Pairs	# Lock Operations
db	Database application.	85277	449	439	1744	22141
Example Vector int	Example in paper.	63	18	18	17	10
java_cup on hwk	Parser generator.	551768	761	761	2948	57774
java_cup on java11	Parser generator.	9353838	753	753	2885	574686
jess	Expert system.	599477	850	830	2900	86947
JLex on hwk	Lexical scanner generator.	9191528	243	240	1115	2271197
JLex on sample	Lexical scanner generator.	3807044	242	239	1104	1839304
mtrt	Two-threaded ray tracer.	5721456	582	572	2712	350574
slice	Radiology data viewer.	1847615	1468	1468	3643	26395
volano	Chat room simulation.	9394365	433	433	841	5021842

Table 3. Characteristics of ten executions.

is the number of bindings that trigger the interprocedural propagations. We include the last column, which gives the number of lock acquisitions of each program,[1] to suggest the effectiveness of synchronization elimination. We found no straightforward measure of the effectiveness of stack allocation, other than the count of optimizable objects.

Ruf demonstrated that a number of applications can be analyzed off-line in a matter of seconds.[10] For example, his analysis of *java_cup* finished in 1.01 seconds and *JLex* in 0.56 seconds. Extremely large applications took about twenty seconds to inspect. An on-line analysis has the advantages of seeing a more precise call graph but has the disadvantage of being incremental.

4.3 Immediate Propagations

The first strategy we discuss is one that starts the analysis when the program starts and analyzes all call site/target bindings immediately. By analyzing the target method before it executes, the analysis has the potential to optimize all objects created by the program. Before an allocation site executes, the analysis will have already analyzed the context of the instruction. Once the run-time system has performed an optimization, the analysis must continue to analyze new targets immediately; otherwise, an unanalyzed binding may cause optimized code to execute incorrectly.

The ability to catch new call site targets is straightforward. If the JVM interprets the call site, it triggers the analysis when the target is previously unseen. If native code is executing the call site, the code triggers the analysis after determining the target but before branching to it. To avoid triggering the analysis on every invocation, one may use a polymorphic inline cache and move the triggering to the fallback of the conditional, as follows:

```
if target is A
   jump directly to A
else if target is B
   jump directly to B
else
   execute method lookup code
   trigger the analysis
   jump directly to correct method
```

This example, which assumes that the analysis has already seen targets A and B, activates the analysis only when the uncommon target arises.

Table 4 illustrates the cost and effectiveness of starting the interprocedural analysis immediately. The second column gives the number of times the analysis propagates a method's summary to a call site. It does not include the unification of arguments for methods within an SCC. This number, which ranges between 2.0 and 3.3 times greater than the number of distinct call site/target bindings, ultimately governs the overhead of the analysis.

The next five columns reveal the types of objects allocated. To determine the type, we look up the classification of the node corresponding to the allocation site and find the most recent method on the call stack in which the corresponding node cannot escape. This allows us to characterize objects that leave factory-type methods with respect to the calling contexts. Of the potential thread-local objects, nearly all objects are classified as *virtually local*. This corresponds to a typical object-oriented program's high use of virtual methods. The most frequently allocated *local* object is the 12-byte array in toString of class Integer. Because of

[1] We did not count entry into synchronized static methods.

Benchmark	# Propa-gations	# Objs. Allocated					# Objs. Optimized		# Objs. Deoptimized		# Lock Ops Eliminated	
		Local	Nonvirt Local	Virt. Local	Shared	Un-known	Opt.	Pes.	Opt.	Pes.	Opt.	Pes.
db	4828	51 (1%)	59 (1%)	2532 (53%)	2133 (45%)	0	2642 (55%)	51 (1%)	901 (34%)	0	12816 (58%)	0
Example	34	0	0	11 (100%)	0	0	11 (100%)	0	0	0	10 (100%)	0
java_cup on java11	9384	32007 (5%)	65 (0%)	127521 (20%)	487226 (75%)	0	159593 (25%)	32007 (5%)	654 (0%)	0	228227 (40%)	0
jess	9667	207 (0%)	404 (1%)	17025 (36%)	19065 (40%)	10914 (23%)	17636 (37%)	207 (0%)	210 (1%)	0	50556 (58%)	0
JLex on sample	3465	440 (1%)	178 (0%)	46165 (97%)	980 (2%)	0	46783 (98%)	440 (1%)	10 (0%)	0	1838356 (100%)	0
mtrt	7326	11038 (4%)	125 (0%)	273561 (91%)	14625 (5%)	0	284724 (95%)	11038 (4%)	100 (0%)	0	349412 (100%)	0
slice	10174	2366 (1%)	578 (0%)	452615 (96%)	13890 (3%)	0	455559 (97%)	2366 (1%)	700 (0%)	0	16626 (63%)	28 (0%)
volano	2269	45727 (5%)	625 (0%)	781332 (93%)	8469 (1%)	0	827684 (99%)	45727 (5%)	71 (0%)	0	5015018 (100%)	0

Table 4. Results of an analysis that begins immediately.

our implementation, we were unable to determine the types of a handful of objects in *jess*.

The remaining columns compare the optimistic and pessimistic optimization approaches, commencing with the number of optimizable objects. For all benchmarks, the number of objects in the pessimistic case is significantly smaller than in the optimistic case. This follows because the analysis does not classify many objects as *local*. Objects tend to flow into virtual methods, and programs do not always execute every call site.

The next comparison is on the number of objects that would need to be deoptimized as a result of marking a node *shared*. The pessimistic approach never needs to deoptimize since, by definition, it does not optimize an object unless it is guaranteed to be thread-local. The optimistic approach, on the other hand, will need to deoptimize, although the figures are much lower than the number of objects optimized. Aside from *db*, it needs to undo between 0% and 1% of its optimizations. *Db* is the exception. Even though the analysis demotes only 27 nodes (1%) to *shared* status, these demotions invalidate 901 optimized objects (34%).

The last comparison is on the number of lock acquisitions that can be removed. The optimistic approach has mixed success; it can remove nearly all lock acquisitions in *JLex* but less than half in *java_cup*. In comparison, the pessimistic approach tends not to optimize objects that are later locked. Only for *slice* does the pessimistic approach remove any locking. Because the pessimistic approach does not prove effective for synchronization elimination on these applications, we disregard it in future tables.

4.4 Delayed Propagations

The efficiency of the static whole-program analysis stems from the fact that it analyzes a binding only once. The incremental version, on the other hand, analyzes a binding repeatedly, as the call graph grows and call sites need updating. This behavior fights the natural progression of program execution.

Nothing requires the analysis to start immediately. To counter the high number of propagations, we can defer the analysis until the program has established a sufficiently large call graph. At this point, the analysis can analyze the SCCs of the current call graph in reverse topological order, thereby reducing the initial number of propagations. After an optimization occurs, however, the analysis must begin an immediate style in order to identify *shared* nodes before the program incorrectly executes.

Since the run-time system may start the analysis at any time, it may be able to hide the analysis behind I/O, time-consuming memory accesses, or garbage collection. The initial interprocedural phase does not

Benchmark	# Meth. Before Prop.	# Propa-gations	# Objs. Allocated					# Objs. Opt.	# Objs. Deopt.	# Lock Ops Elim.
			Local	Nonvirt Local	Virtually Local	Shared	Un-known			
db	430	732 (↓ 85%)	11 (0%)	9 (0%)	446 (9%)	33 (1%)	4276 (90%)	466 (9%)	14 (3%)	4197 (19%)
java_cup on java11	190	8046 (↓ 14%)	31907 (5%)	45 (0%)	117445 (18%)	485906 (75%)	11516 (2%)	149397 (23%)	353 (0%)	127737 (22%)
jess	708	5552 (↓ 43%)	177 (0%)	69 (0%)	10835 (23%)	8406 (18%)	28127 (59%)	11082 (23%)	61 (1%)	25498 (29%)
JLex on sample	193	2103 (↓ 39%)	439 (1%)	17 (0%)	44022 (92%)	963 (2%)	2322 (5%)	44478 (93%)	3 (0%)	1686615 (92%)
mtrt	391	4266 (↓ 42%)	10937 (4%)	34 (0%)	102274 (34%)	12777 (4%)	173327 (58%)	113245 (38%)	43 (0%)	3753 (1%)
slice	404	8660 (↓ 15%)	2344 (1%)	512 (0%)	450976 (96%)	13018 (3%)	2255 (0%)	453832 (97%)	153 (0%)	11606 (44%)
volano	318	1817 (↓ 20%)	54612 (6%)	591 (0%)	779255 (92%)	8058 (1%)	2637 (0%)	825458 (98%)	66 (0%)	5007397 (100%)

Table 5. Results of an analysis that begins after 50,000 instrumentation ticks of no new methods.

need to complete before the program resumes; it may be interspersed.

When should a delayed analysis start? If it waits too long, it will miss chances for optimization. In the worst case, an application allocates all optimizable objects before the analysis begins. If it starts too early, it will face the same number of propagations as the immediate approach. As a compromise, a delayed analysis could start when the rate of class loading slows, when the rate at which new methods execute slows, or when the run-time system detects a frequently executed portion of the application. In any case, one can devise an application that countermines the chosen delay strategy.

We ran an experiment in which the analysis started after 50,000 instrumentation ticks had occurred without causing a new method to be executed. We felt that this number would give a program ample time to settle down. The results appear in Table 5.

The second column of the table lists the number of methods executed before the analysis starts. The number of propagations, listed in the third column, is smaller than in the immediate case, reducing it on average by 37%. In general, however, additions to the call graph at the end of the program often require more propagations than additions at the beginning, causing the bulk of the propagations to remain in the delayed approach.

The distribution of the types of objects allocated is nearly identical to the previous approach. Unknowns result when programs allocate objects before the initial analysis has run. These potentially translate into missed opportunities for optimization. For example, the delayed analysis eliminates about 50% fewer lock acquisitions for *jess*.

4.5 Persistent Propagations

A viable strategy must have, in the common case, a low run-time overhead and a high potential for optimization. We have seen that an immediate analysis has the potential to discover all optimizable objects but may incur high propagation costs. We have also seen that delaying an initial analysis can reduce the number of propagations but may miss optimizations.

We ran a third experiment in which the analysis utilized previous results on a given application. Table 6 shows measurements of two scenarios. The first analyzes *JLex* on *hwk*, using the call graph and results of *JLex* on *sample*. Even though the *hwk* input file is much larger than *sample*, the analysis only analyzes 2 additional methods and 18 additional bindings. This causes 63 propagations—a huge drop from 3465 of the first input. Moreover, the run-time system can optimize a high percentage of objects and does not need to undo any of the optimizations.

The second scenario analyzes *java_cup* on *hwk* using the results of *java_cup* on *java11*. This time *hwk* is a much smaller input than the first input file. The number of previously unseen methods and bindings again decreases significantly, incurring fewer than 200 propagations.

Benchmark	# Meth. Ana-lyzed	# Invoc. Ana-lyzed	# Propa-gations	# Objs. Allocated				# Objs. Opti-mized	# Objs. Deopti-mized	# Lock Ops Elim.
				Local	Nonvirt. Local	Virtually Local	Shared			
1. JLex on sample	242	1104	3465	440 (1%)	178 (0%)	46165 (97%)	980 (2%)	46783 (98%)	10 (0%)	1838356 (100%)
2. JLex on hwk	2	18	63	2241 (0%)	313 (0%)	472609 (98%)	4713 (1%)	475163 (99%)	0 (0%)	2269005 (100%)
1. java_cup on java11	753	2885	9384	32007 (5%)	65 (0%)	127521 (20%)	487726 (75%)	159593 (25%)	654 (0%)	228227 (40%)
2. java_cup on hwk	8	89	190	4318 (9%)	29 (0%)	10992 (22%)	34402 (69%)	15339 (31%)	1 (0%)	39809 (69%)

Table 6. Results of an analysis that utilizes previous results.

Call sites typically target the same methods on subsequent executions. By reusing results, this approach avoids analyzing a binding repeatedly and skirts deoptimization. If the dynamic properties of an execution change (for example, the input changes or the classpath variable is modified, leading to a different binding at a nonvirtual call site), the analysis will catch the changes and take the union of the previous and new scenarios. The result, although less precise, is still safe.

This approach still has several disadvantages. First, a *shared* node will persist in all subsequent executions even if these executions do not let the node escape the thread. For example, suppose one execution of our Example program uses a List data structure that places itself in a static field. The analysis will mark the list *shared* and on subsequent executions will never consider optimizing the list. Second, the analysis may miss the (possibly rare) situation where it is better to optimize an object and later deoptimize it than to disqualify it from optimization.

The results of Section 4 seem to suggest the following optimization strategy. On the first execution of a program, perform no analysis while the program executes but record the binding of call sites to target methods. This sacrifices optimizations in the current execution but incurs little overhead. After the program finishes, use an efficient whole-program version to analyze the entire program and then save the results. On subsequent executions use the saved results and an optimistic strategy to optimize the program.

5. Related Work

Shape analysis has recently entered the scene to identify optimizable objects in Java. Researchers aim to stack allocate and to remove synchronization on thread-local objects.[1,3,4,5,6,10,16] All of these analyses assume a closed, known world and work in a static compiler. Shape analysis is also being used to help program verification and model checking.[7,9] At this time, however, it is not clear if these latter uses will benefit from a dynamic, incremental approach.

The field analysis by Ghemawat and Randall [8] avoids having to know the entire program by looking at the access flags of fields and methods. For example, members with package scope can be accessed only within the enclosing package. This idea, coupled with the idea of package sealing [18], which restricts all classes within a package to come from the same archive file, may allow us to classify more objects as *local*. For instance, a *virtually local* node may change to *local* if it only flows to sealed call sites and if all potential targets have been analyzed. Our current analysis examines neither access flags nor type information.

The most common dynamic analysis for Java is just-in-time compilation. It trades off compilation overhead for increased execution. Since it is not a whole-program analysis, it can be selective about what it compiles. For example, it may choose to compile only frequently executing methods. If the compilation becomes too expensive, it can fall back to a naïve native code translation or even to interpreted code. A dynamic shape analysis faces a different battle because it cannot simply quit if the analysis becomes too time consuming. If the analysis fails to analyze a method, an incorrect execution may ensue.

Two recent publications move toward dynamic interprocedural analyses. The first, by Sreedhar *et al.* [13], presents a framework called *extant analysis* that, during an off-line static analysis, characterizes all references as either *unconditionally extant* or *conditionally extant*. The former denotes a reference to

Optimization Approach	Benefits	Drawbacks
Optimistic	Has the potential to optimize all objects	May need to deoptimize
Pessimistic	Precludes deoptimization	Guarantees few thread-local objects May miss optimizations

Table 7. Trade-offs regarding the optimization strategy.

Analysis Approach	Benefits	Drawbacks
Immediate	Has the potential to optimize all objects	Faces numerous propagations
Delayed	Reduces the number of propagations	May miss optimizations
Persistent	Infrequently propagates after the first execution	Requires additional start-up and exit costs Inherits worst case over all executions

Table 8. Trade-offs regarding the start of the interprocedural analysis.

an object whose type is guaranteed to be in a specified closed world, and the latter captures the remaining references. In our work, our notion of a closed world gets defined as the program executes. If a reference flows into a virtual method (*i.e.*, its node is *virtually local*), it exits the closed world. Only fully analyzed nodes that remain in the closed world can be guaranteed to be thread-local.

The second, by Serrano *et al.* [11], introduces a quasi-static analysis, named Quicksilver, that saves compiled code between program executions. Before execution, it validates the existing code images and "stitches" them to reflect the current run-time properties. Their approach is effective for the SPECjvm98 benchmarks, once it has processed an initial "learning" execution. This approach is similar to our persistent strategy, although we do not require stitching; we take the union over all previous executions.

6. Conclusions

We adapted an efficient whole-program shape analysis to operate incrementally and on-the-fly. Its dynamic property enables it to observe the dynamic call graph, to analyze only the executing methods, and to propagate information only to executing call sites. Also, the dynamic behavior allows it to perform optimizations not expressible in bytecode.

In general, when a call site targets a new method, the shape analysis propagates information from the target method to all affected parts of the call graph. Table 7 and Table 8 summarize various approaches to performing the analysis and a related optimization. Our results suggest that an optimizer must optimistically select objects to optimize in order to identify a large number of optimization points. For the applications we studied, an optimizer that takes this approach will need to undo only a small fraction of its optimizations. A strategy that propagates immediately is able to optimize objects as soon as possible but incurs a large cost for propagations. To alleviate the propagation cost, we can delay the initial analysis. Doing so, however, does not significantly reduce the number of propagations. We can eliminate most of the propagations if we reuse the analysis results from previous executions.

This paper does not attempt to close the book on the dynamic shape analysis problem; it provides some empirical evidence of the difficulty of performing it dynamically and suggests an approach that may be both effective and attainable. Two related issues must be resolved before it can be used in practice.

First, because any thread of the program may trigger the analysis, it must be thread-safe. Moreover, it must handle concurrent activation without sacrificing efficiency. If the analysis uses union-find data structures to efficiently merge nodes and SCCs, as suggested in [10], work done by Anderson and Woll regarding parallel union-find algorithms [2] may prove helpful.

Second, optimization and deoptimization techniques must be explored. Deoptimization is especially difficult. When the analysis marks a node *shared*, it must deoptimize all objects that are represented by this node. This requires the run-time system to identify all objects allocated as a result of the optimization and possibly to recompile optimized code. In the case of lock elimination, the process not only needs to enable all future lock operations on the deoptimized object but also needs to grant the owning thread the current number of nested lock acquisitions that would have been held had the object not been optimized. In the case of stack allocation, it may need to move an optimized object from the stack to the heap.

We hope that this study encourages JVM implementors to consider whole-program analyses that aid in dynamic optimizations. We are currently looking into a dynamic shape analysis that can be driven by frequently accessed objects instead of by changes to the call graph. Such an approach may be able to eliminate the requirement that the analysis examine the entire program.

Acknowledgments

This work was funded 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*, Venezia, Italy, September 1999.

[2] Richard J. Anderson and Heather Woll. Wait-free Parallel Algorithms for the Union-Find Problem. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 370-380, New Orleans, Louisiana, 5-8 May 1991.

[3] Jeff Bogda. Detecting Read-Only Methods in Java. In *Proceedings of the Fifth International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR '00)*, pages 143-154, Rochester, New York, 25-27 May 2000.

[4] Jeff Bogda and Ambuj Singh. *Critical Section, Be Gone!* Technical Report TRCS00-18, Department of Computer Science, University of California, Santa Barbara, August 2000.

[5] 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.

[6] 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.

[7] 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.

[8] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of Programming Languages, Design, and Implementation (PLDI '00)*, pages 334-344, Vancouver, Canada, 18-21 June 2000.

[9] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting Static Analysis to Work for Verification: A Case Study. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '00)*, pages 26-38, Portland, Oregon, 21-24 August 2000.

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

[11] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quicksilver: A Quasi-Static Compiler for Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 66-82, Minneapolis, Minnesota, 15-19 October 2000.

[12] SPEC Java virtual machine benchmark suite. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation, Release 1.0*. August 1998. http://www.spec.org/osq/jvm98/jvm98/doc/index.html.

[13] Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of Programming Languages, Design, and Implementation (PLDI '00)*, pages 196-207, Vancouver, Canada, 18-21 June 2000.

[14] Frank Tip and Jens Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 281-293, Minneapolis, Minnesota, 15-19 October 2000.

[15] Volano benchmark application. http://www.volano.com/benchmarks.html.

[16] 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.

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

[18] Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. Sealed Calls in Java Packages. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 83-92, Minneapolis, Minnesota, 15-19 October 2000.