RC22340 (W0205-208) May 29, 2002
Computer Science

IBM Research Report

Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis

Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano,
Vugranam C. Sreedhar, Samuel P. Midkiff
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis

JONG-DEOK CHOI, MANISH GUPTA, MAURICIO J. SERRANO, VUGRANAM C. SREEDHAR and SAMUEL P. MIDKIFF

This article presents a framework for *escape analysis* for Java to determine (1) if an object is not reachable after its method of creation returns, so that the object can be allocated on the stack, and (2) if an object is reachable only from a single thread during its lifetime, allowing unnecessary synchronization operations on that object to be removed. We introduce a new program abstraction for escape analysis, the *connection graph*, that is used to establish reachability relationships between objects and object references. We show that the connection graph can be succinctly summarized for each method such that the same summary information may be used in different calling contexts without introducing imprecision into the analysis. We present an interprocedural algorithm that uses the above property to efficiently compute the connection graph and identify the non-escaping objects for methods and threads. We prove the correctness of this algorithm. Finally, we describe additional analysis (not yet incorporated in our current implementation) that can be used to eliminate redundant storage synchronization operations associated with locks in Java. The experimental results, from a prototype implementation of our framework in the IBM High Performance Compiler for Java, are very promising. The percentage of objects that may be allocated on the stack exceeds 70% of all dynamically created objects in the user code in three out of the ten benchmarks (with a median of 19%), 11% to 92% of all mutex lock operations on objects created in user code are eliminated in those ten programs (with a median of 51%), and the overall execution time reduction ranges from 2% to 23% (with a median of 7%) on a 333 MHz PowerPC workstation with 128 MB memory.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*; E.1 [**Data**]: Data Structures—*graphs; trees*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Connection graphs, escape analysis, points-to graph

1. INTRODUCTION

Java has become an important language for general-purpose computing and for server applications. Performance is an important issue in these application environments. In Java, each object is conceptually allocated on the heap and can be deallocated only by garbage collection. Also, each object has a lock associated with it, which is used to ensure mutual exclusion when a synchronized method or statement is invoked on the object. In this paper we present a technique for identifying objects that are local to a method invocation and/or local to a thread. Once we identify those local objects we can perform two important optimizations for Java programs.

(1) If an object is local to a method invocation it can be allocated on the method's

Authors' address: IBM T. J. Watson Research Center; P. O. Box 218, Yorktown Heights, NY 10598; {jdchoi, mgupta, mserrano, vugranam, smidkiff}@us.ibm.com.
A preliminary version of this paper appears in the proceedings of the OOPSLA '99 Conference.

stack frame. Stack allocation reduces garbage collection overhead, since the storage on the stack is automatically reclaimed when the method returns (although unconstrained stack allocation can lead to additional memory pressure, since the stack frame itself is not garbage collected). Also, by allocating objects on the local stack, we reduce the occasional synchronization that the heap allocator has to do with other threads competing for memory chunks. Besides stack allocation, the knowledge about an object being local to a method can enable further optimizations, such as, more aggressive code reordering in spite of the *precise exception* semantics of Java [Gosling et al. 1996] by allowing writes of method-local variables to be moved across potentially excepting instructions in a method without any exception handler [Chambers et al. 1999]. As well, with further analysis, the object accesses can be strength-reduced, and the creation of the object may be eliminated.

(2) If an object is local to a thread, then no other thread can access the object. This has several benefits, especially in a multithreaded multiprocessor environment. First, we can eliminate the synchronization operations that ensure mutual exclusion on this object. Note that Java memory model still requires that we flush the Java local memory at `monitorenter` and `monitorexit` statements in bytecode (inserted for synchronized statements and method calls). Second, objects that are local to a thread can be allocated to improve data locality. Third, with further analysis that we shall describe, some operations to flush the local memory can be safely eliminated. Finally, more aggressive code reordering optimizations that move writes across potentially excepting instructions (similar to those described above, but for the case when there is no user-defined exception handler for the given thread) can be enabled by identifying thread-local objects [Gupta et al. 2000].

The term *escape analysis* has been used in the literature [Park and Goldberg 1992] for the analysis to determine the set of the objects that escape a method invocation. If an object escapes a method invocation (thread), we say it is not local to that method invocation (thread). We introduce a framework for escape analysis, based on a simple program abstraction called the *connection graph*. The connection graph abstraction captures the "connectivity" relationships among heap allocated objects and object references. For escape analysis, we perform reachability analysis on the connection graph to determine if an object is local to a method or local to a thread. Different variants of our analysis can be used either in a static Java compiler, a dynamic Java compiler, a Java application extractor, or a bytecode optimizer. To evaluate the effectiveness of our approach, we have implemented various flavors of escape analysis in the context of a static Java compiler [International Business Machines Corporation 1997; Seshadri 1997], and have analyzed ten medium to large benchmarks.

The main contributions of this paper are:

—We present a new, simple interprocedural framework (with flow-sensitive and flow-insensitive versions) for escape analysis in the context of Java.

—We demonstrate an important application of escape analysis for Java programs – that of eliminating unnecessary lock operations on thread-local objects. It leads to significant performance benefits even when using a highly optimized

implementation of locks, namely, *thin-locks* [Bacon et al. 1998]. We also describe additional analysis that allows redundant memory update and flush operations associated with Java locks to be eliminated.

—We describe how to handle *exceptions* in the context of escape analysis for Java, without being unduly conservative. These ideas can be applied to other data flow analyses in the presence of exceptions as well.

—We introduce a simple program abstraction called the connection graph, which is well suited for the purpose of escape analysis. It is different from points-to graphs for alias analysis whose major purpose is memory disambiguation. In the connection graph abstraction, we also introduce the notion of *upwards-exposed objects* and *phantom nodes*, which allow us to summarize the effects of a callee procedure independent of the calling context. This succinct summarization helps improve the overall speed of the algorithm.

—We present extensive experimental results from an implementation of escape analysis in a Java compiler. We show that in the user code (not including the class libraries), the compiler is able to detect more than 19% of dynamically created objects as stack-allocatable in five of the ten benchmarks that we examined (finding higher than 70% stack-allocatable objects in three programs). We are able to eliminate 11%-92% of lock operations (that ensure mutual exclusion) on the objects created in the user code in those ten programs. The overall performance improvement ranges from 2% to 23% on a 333 MHz IBM PowerPC workstation with 128 MB memory.

The rest of this paper is organized as follows. Section 2 presents our connection graph abstraction. Section 2.3 formalizes the notion of escape of objects. Sections 3 and 4 respectively describe the intraprocedural and interprocedural analyses, to build the connection graph and to identify the objects that do not escape their method or thread of creation. Section 5 elaborates on our handling of special Java features like exceptions and object finalizers. Section 6 describes an extension (not yet incorporated in our current implementation) to detect cases in which the storage synchronization operations associated with Java locks can be eliminated. Section 7 describes the transformation and the run-time support for the optimization, and Section 8 presents experimental results. Section 9 discusses related work, and Section 10 presents conclusions. Appendix A proves the correctness of our algorithm, and Appendix B derives the complexity of our algorithm.

2. CONNECTION GRAPH REPRESENTATION FOR ESCAPE ANALYSIS

In Java, objects are run-time instances of classes. Run-time objects in Java are created via `new` statements[1] and objects are referenced by *object references*. A run-time object is composed of a collection of named fields. A field can be either a reference to another object or a non-reference (non-pointer) value. During escape analysis, we abstract the run-time objects and represent them as compile-time objects. In this article, we use the term *concrete objects* for run-time objects and the term *abstract objects* for compile-time objects.

[1]In Java, a cloning site is also an allocation site, and is handled similarly by our analysis.

2.1 Access Paths

Access paths are used to represent reachability relationships among concrete objects. We formalize this as follows:

Definition 2.1. An *access path*, $p.\alpha$, is a pair consisting of a reference variable p and a sequence of field reference names, $\alpha = f_1.f_2 \dots f_n$.

Definition 2.2. The destination of an access path $dest(p.f_1.f_2 \dots f_n)$ is the concrete object referenced by the field reference f_n.

Note that access paths, a term we use for a concrete graph, should not be confused with compile-time reference expressions, a term we use for an abstract graph.

Definition 2.3. A concrete object O' is said to be reachable from another concrete object O if there exists an access path $p.f_1.f_2 \dots f_n$ $(n \geq 0)$ such that $dest(p.f_1.f_2 \dots f_i) = O, dest(p.f_1.f_2 \dots f_j) = O'$, and $0 \leq i \leq j \leq n$.

We shall use the notion of an access path and its static abstraction in developing our escape analysis algorithm and in proving the correctness of the algorithm.

2.2 Connection Graph

A Connection Graph (CG) is a finite, labeled, directed graph, which has nodes representing (abstract) objects, fields, and object references, and edges representing reachability relationships among them. In our framework, we use a 1-limited naming scheme for objects, which creates one abstract object for each allocation statement and at most one abstract object (called a *phantom object*) for each non-allocation statement in the program.

Definition 2.4. A *connection graph* is a directed graph $CG = (N_o \cup N_r \cup N_a \cup N_f \cup N_g, E_p \cup E_d \cup E_f)$, where

—N_o represents the set of abstract objects.

—N_r represents the set of reference variables (those locals and formals that are object references) in the program.

—N_a represents the set of actual parameters, including return values, which are object references. Actuals are implicit in the program and the nodes corresponding to actuals are constructed during interprocedural analysis.

—N_f represents the set of non-static fields that are object references. These are called *field reference nodes*.

—N_g represents the set of static fields, i.e., all *global variables*, that are object references. (All nodes in the set $N_r \cup N_f \cup N_a \cup N_g$ are collectively referred to as the *reference nodes*.)

—E_p is the set of *points-to* edges. A points-to edge exists from a reference node r to an object node o if the object reference corresponding to r may point to the object corresponding to o. If $p \rightarrow q \in E_p$, then $p \in N_r \cup N_a \cup N_f \cup N_g$ and $q \in N_o$.

—E_d is the set of *deferred edges*. A deferred edge from a node p to a node q signifies that p points to whatever q points to. If $p \rightarrow q \in E_d$, then $p, q \in N_r \cup N_a \cup N_f \cup N_g$.

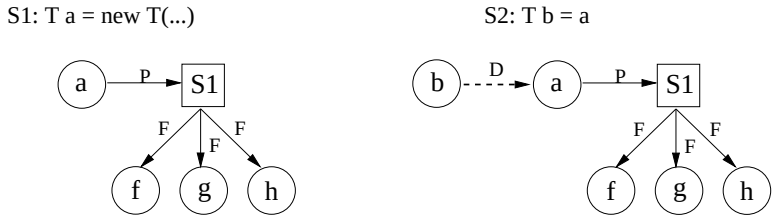


Fig. 1. A simple connection graph. Boxes indicate object nodes and circles indicate reference nodes (including field reference nodes). Solid edges indicate points-to edge, dashed edges indicate deferred edges, and edges from boxes to circles indicate field edges.

—E_f is the set of *field edges*. A field edge from o to f signifies that f represents a field of object o. If $p \rightarrow q \in E_f$, then $p \in N_o$ and $q \in N_f \cup N_g$.

We handle arrays in Java like other objects. Conceptually, an array object contains a number of variables, called the *components* of the array. The components of the array are referenced via integer-typed indices from 0 to $n-1$ inclusively, where n is the number of the components (or size) of the array. In our analysis, we do not apply any array-index analysis: we do not distinguish between different components of the same array.

Figure 1 illustrates an example of a connection graph. In figures, we represent each abstract object as a tree with the root representing the object and the children of the root representing the reference fields within the object.[2] Also, in our figures, a solid-line edge represents a points-to edge or a field edge, and a dashed-line edge represents a deferred edge. In the text, we use the notation $p \xrightarrow{P} q$ to represent a points-to edge from node p to node q, $p \xrightarrow{D} q$ to represent a deferred edge from p to q, and $p \xrightarrow{F} q$ to represent a field edge from p to q.

Given an object node O_a, we define a mapping $AllocSite(O_a)$ that returns the program point (i.e., allocation site) where O_a is created. We define a mapping $ObjMap : O_a \rightarrow \mathcal{O}_c$ from an abstract object O_a to a set of concrete objects $\mathcal{O}_c$. The range $\mathcal{O}_c = ObjMap(O_a)$ is the set of all concrete objects allocated at the site $AllocSite(O_a)$.

Given a reference node $p \in N_r \cup N_a \cup N_f \cup N_g$, the set of abstract object nodes $\mathcal{O} \subseteq N_o$ that it (immediately) points-to can be determined by traversing the deferred edges from p until we visit the first points-to edge in the path. The destination node of the points-to edge will be in $\mathcal{O}$. This generalizes to the notion of points-to path as follows:

Definition 2.5. Let $p \in N_r \cup N_a \cup N_f \cup N_g$. A points-to path of length n, denoted as $p \xrightarrow{+n} q$, is a sequence of edges $p = p_0 \rightarrow p_1 \rightarrow \ldots \xrightarrow{P} q$ that terminates in a points-to edge and contains exactly n points-to edges in the path.

[2]Since Java does not allow nested objects, the tree representation of an object consists of only two levels – the root and its children.

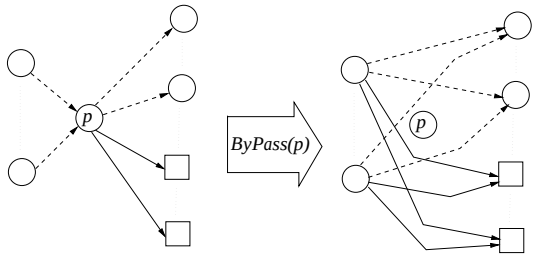


Fig. 2. Illustrating $ByPass(p)$ function

Definition 2.6. Given a points-to path of length n, $p \xrightarrow{+n} q$, we define

$PointsTo(p \xrightarrow{+n} q) = \{q | q$ is the destination node of the last edge in the path$\}$.

We use the shorthand notation *PointsTo(p)* to refer to the nodes immediately pointed to by p:

Definition 2.7. Let $p \in N_r \cup N_a \cup N_f \cup N_g$, then the set of object nodes that nodes p points-to is:

$$PointsTo(p) = \{q | p \xrightarrow{+1} q\}.$$

We use deferred edges to model assignments that merely copy references from one variable to another. Deferred edges defer computations during connection graph construction, and thereby help in reducing the number of graph updates needed during escape analysis. Deferred edges were first introduced for flow-insensitive pointer analysis in [Burke et al. 1995]. One can always eliminate deferred edges by redirecting incoming deferred edges to the successor nodes. In other words, we define a bypass function $ByPass(p)$ that when applied to a reference node p redirects the incoming deferred edges of p to the successor nodes of p. The type of redirected edge is the same as the type of edge from p to the corresponding successor node. It also removes any outgoing edges from p. Figure 2 illustrates the $ByPass(p)$ function. More formally, let $R = \{r | r \xrightarrow{D} p\}$, $S = \{s | p \xrightarrow{P} s\}$, and $T = \{t | p \xrightarrow{D} t\}$. $ByPass(p)$ removes the edges in the set $\{r \xrightarrow{D} p\} \cup \{p \xrightarrow{P} s\} \cup \{p \xrightarrow{D} t\}$ from the connection graph (CG) and adds edges in the set $\{r \xrightarrow{P} s | r \in R$ **and** $s \in S\} \cup \{r \xrightarrow{D} t | r \in R$ **and** $t \in T\}$ to the CG. Note that $ByPass(p)$ can always be applied to a reference node to eliminate its incoming deferred edges.

2.3 Escape Property of Objects

We now formalize the notion of *escape* of an object from a method or a thread.

Definition 2.8. Let O be a concrete object and M be a method invocation. O is said to escape M, denoted as $Escapes(O, M)$, if the lifetime of O may exceed the lifetime of M.

Definition 2.9. Let O be a concrete object and T be a thread (instance). O is said to escape T, again denoted as $Escapes(O, T)$, if O is visible to another thread $T' \neq T$.

Alternatively, we say that a concrete object O is *stack-allocatable* in M if $\neg Escapes(O, M)$, and that a concrete object O is *local* to a thread T if $\neg Escapes(O, T)$.

Let M be a method invocation in a thread T. The lifetime of M is, in that case, bounded by the lifetime of T. If another thread object, T', is created in M, we conservatively set $Escapes(O', M)$ to be true for all objects O' (including T') that are reachable from T' (since the thread corresponding to T' may continue executing even after M returns). Thus, we ensure that a concrete object, whose lifetime is inferred by our analysis to be bounded by the lifetime of a method, can only be accessed by a single thread.

For static escape analysis, we map concrete objects to nodes in the CG, and access paths to paths in the CG. For each CG node, we can identify an allocation site in the program where the concrete object instances are created for that CG node. Each node in CG has an *escape state* associated with it. For marking escape states, we define an escape lattice consisting of three elements: *NoEscape* ($\top$), *ArgEscape*, and *GlobalEscape* ($\bot$). The ordering among the lattice elements is: *GlobalEscape* < *ArgEscape* < *NoEscape*. *NoEscape* means that the object does not escape the method in which it was created. *ArgEscape* with respect to a method means that the object escapes that method via the method arguments or return value, but does not escape the thread in which it is created. Finally, *GlobalEscape* means that the object is regarded as escaping globally (i.e., all methods and its thread of creation). Let $EscapeSet = \{NoEscape, ArgEscape, GlobalEscape\}$, and let $es \in EscapeSet$. We define the escape state merge as follows:

$$\begin{aligned} es \wedge es &= es \\ es \wedge NoEscape &= es \\ es \wedge GlobalEscape &= GlobalEscape \end{aligned}$$

We conservatively assume that each static field and thread object[3] outlives every method in the program. Hence, we initialize the escape state of nodes representing static fields (i.e., nodes in N_g) and thread objects to *GlobalEscape*. As discussed in Section 4, the escape state of placeholder nodes representing actual parameters of a method is initialized to *ArgEscape*. All other nodes are marked *NoEscape* initially.

In order to compute the escape state of an abstract object, we perform reachability analysis on the CG. Consider an object node $o = PointsTo(p)$ and a field node q of o (i.e., $o \xrightarrow{F} q$). We ensure, using our analysis, that:

$$\begin{aligned} EscapeState(o) &\leq EscapeState(p) \\ EscapeState(q) &= EscapeState(o) \end{aligned}$$

Note that even though a thread object (which can be accessed in both the creating and the created thread) and all objects reachable from it are marked *GlobalEscape*, it does not mean that all objects created *during* the execution of a thread will be marked *GlobalEscape*.

[3]We regard any run-time instance of a class that implements the `Runnable` interface as a thread object.

At the completion of escape analysis, all concrete objects that are allocated at an allocation site whose escape state is marked *NoEscape* are stack-allocatable in the method in which they are created. Furthermore, all concrete objects that are allocated at an allocation site whose escape state is marked *NoEscape* or *ArgEscape*, are local to the thread in which they are created, and so we can eliminate the synchronization in accessing these concrete objects without violating Java semantics.

2.4 Connection Graph versus Points-To Graph

The connection graph has some similarities to, but also a few important differences with, the *points-to* graph representation that has been proposed in the literature for pointer analysis [Sagiv et al. 1998]. (The *compact alias set* [Choi et al. 1993] is also equivalent to the points-to graph representation in the context of Java.) Unlike a points-to graph, a connection graph does not approximate the shape of the concrete storage, but approximates the relevant paths in the concrete storage. For every path from a variable to an object in the concrete storage, there exists a corresponding path in the connection graph from a reference node with the same or lower escape state to that object.

A major difference between points-to graphs such as those used in [Choi et al. 1993; Sagiv et al. 1998] and our connection graph is that the former in general have the following *uniqueness property* that our connection graph does not:

Property (Uniqueness Property). *Let G^c be the concrete (dynamic) storage graph during execution of a program and G^a be an abstract (static) graph representing G^c. Then, a concrete object O_c in G^c has a unique abstract object O_a in G^a that O_c maps to.*

A concrete object, however, can be mapped to multiple abstract objects in the connection graph. Thus, the connection graph lacks the uniqueness property. The connection graph, however, can still be used for computing certain static properties of the program such as computation of escaping objects. We now describe an example program segment that illustrates how this non-uniqueness arises and how that does not affect the computation of escaping objects.

Figure 3 shows a program segment to illustrate the connection graph construction and the difference between the points-to graph and the connection graph. Figure 4 shows the points-to graph and the connection graph at various points of the program in Figure 3. Abstract objects are named after their allocation site. Figure 4(A) shows the points-to graph holding at the entry of method `T()`. Static variables `x` and `y`, and formal parameters `f1` and `f2` of `T()` all point to object `S0`.

Figure 4(B) shows the points-to graph holding after statement `S4`, with two additional objects `S3` and `S4` created at statements `S3` and `S4`, respectively. From the points-to graph in Figures 4(B), we can identify data dependences from statement `S5` to statement `S6` because `f1.data` and `f2.data` refer to the field `data` of the same object (`S0`). Figure 4(C) shows the points-to graph holding after statement `S7`. Figure 4(D) shows the connection graph holding immediately after `S4`. The upper subgraph of the figure shows the connection graph built after statement `S3`, and the lower subgraph shows the connection graph built after statement `S4`. Consider the upper graph. We construct connection graphs in a bottom-up fashion: from callees to callers. Therefore, at the entry of method `T()`, we do not have the

```
class ListElement {

   int data;
   ListElement u, l;
   static ListElement x, y;

   static void L() {
S0:   x = new ListElement();
S1:   y = x;
S2:   T(x,y);
      . . .
   }

   static void T(ListElement f1, ListElement f2) {
S3:   f1.u = new ListElement();
S4:   f2.l = new ListElement();
S5:   f1.data = 10;
S6:   f2.data = f2.data + 20;
S7:   f2 = new ListElement();
  }
}
```

Fig. 3. An example program for illustrating connection graph.

connection graph holding at the callers of `T()`, nor is it necessary. When `f1.u` is write accessed at `S3`, `f1` does not point to any object. However, `f1` still has the same value as the first actual parameter passed by callers of `T()`. We use `a1` to represent the value(s) of the actual parameter passed to formal parameter `f1`. Note that formal parameters such as `f1` and `f2` are local variables of `T()`, but that actual parameters such as `a1` and `a2` are accessible from callers of `T()` and are regarded as non-local variables of `T()`.

We first insert deferred edge from `f1` to `a1`. This edge denotes that `f1` points to what `a1` points to, *not* that `f1` points to `a1`. Deferred edges are deleted once the connection graph of a method is constructed, using a *bypass function* described earlier. We then create a *phantom node*, labeled `S3*`, to represent the object that `a1` points to in a caller's context and whose `u` field is accessed through `f1` at statement `S3`. Similarly, node `S4*` is the phantom node created for the object accessed at `S4`. We also call the actual parameters, such as `a1` and `a2`, the *phantom reference nodes*. The lower subgraph of Figure 4(E) is built similarly after statement `S4`. Note that a single concrete object created at statement `S0` is mapped to two abstract objects `S3*` and `S4*` in `T()`, illustrating the lack of the uniqueness property of the connection graph.

From the connection graph holding after `S4` (Figure 4(D)), we would fail to identify that `f1.data` accessed at `S5` and `f2.data` accessed at `S6` refer to the same storage location. This failure would be fatal for points-to graphs built for the purpose of alias analysis. This failure, however, is not a concern for escape analysis: our escape analysis still identifies that objects `S3` and `S4` are reachable from outside `T()` through actual parameters `a1` and `a2`, respectively. After statement `S7`, `f2` no longer points to object `S4`, as shown in Figure 4(E). Object `S4`, however, is

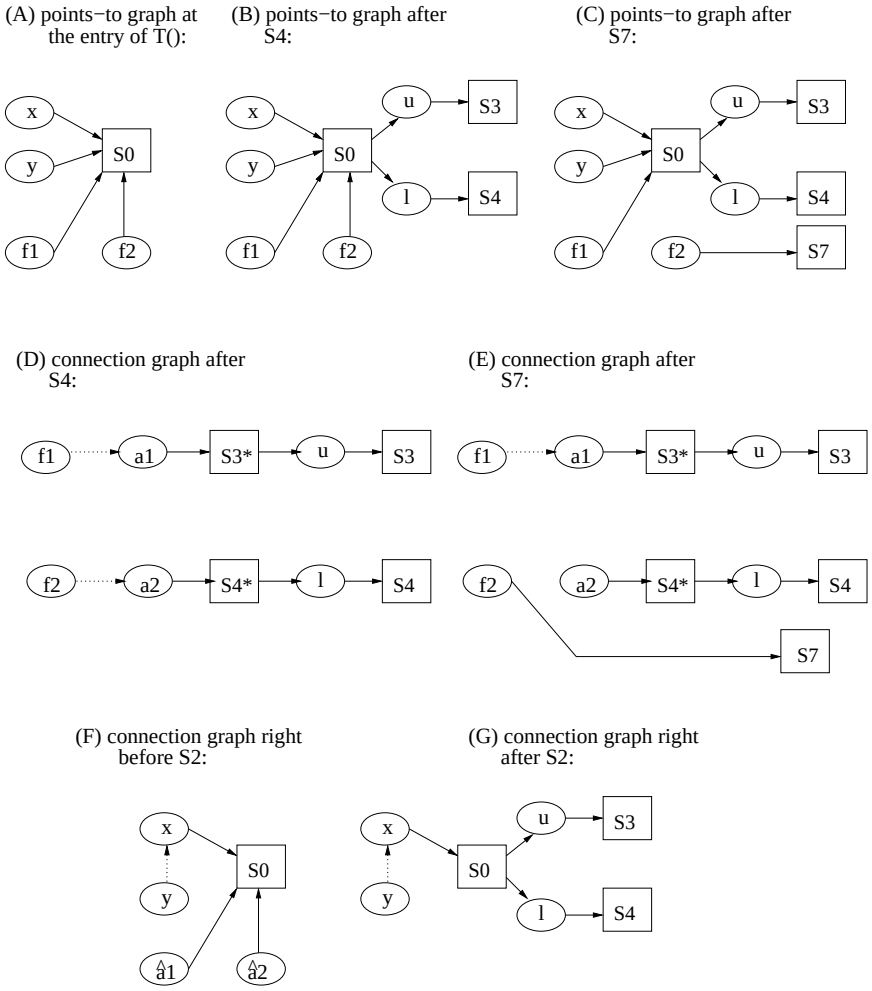


Fig. 4. Points-to and connection graphs at various points in the example program of Figure 3.

reachable from `a2`, making it escape the method (`T()`) in which it is created. Object `S7` is reachable only from a local variable (`f2`), and is identified as a method-local object. The summary connection graph of method `T()` consists of the subgraph of the connection graph after `S7` that is reachable from any non-locals of `T()`: the subgraph reachable from `a1` and `a2` in Figure 4(E).

Now consider method `L()`. The connection graph holding immediately before the call to `T()` at statement `S2`, shown in Figure 4(F), is the same as the points-to graph in Figure 4(A) except for the following: instead of nodes labeled `f1` and `f2`, we have two nodes labeled $\hat{a}$`1` and $\hat{a}$`2`, respectively, pointing to the object labeled `S0`. Nodes labeled $\hat{a}$`1` and $\hat{a}$`2` denote the actual parameter values passed to the call to `T()`, and correspond to the nodes labeled `a1` and `a2`, respectively, in the callee's connection graph. At statement `S2`, we update the connection graph holding immediately

before the call to `T()` (in Figure 4(F)) with the summary connection graph of `T()`, by recognizing that node $\hat{a}$`1` at `S2` corresponds to node `a1` at `S7`, and that node $\hat{a}$`2` at `S2` corresponds to node `a2` at `S7`. From this we can recognize that object nodes `S3*` and `S4*` represent the same object node `S0` at statement `S2`, resulting in the connection graph in Figure 4(G). This process of updating the connection graph holding immediately before a callsite with the summary connection graph of the called method is performed by a routine called `UpdateCaller()`, described later in Section 4. The summary connection graph of a method is constructed independently of the calling context. The effect of the summary connection graph of a method, however, is reflected to each callsite specifically depending on the connection graph holding at each callsite.

To summarize, here are some of the characteristics of our connection graph representation: (1) we use deferred edges to defer computation, and later use a bypass function to evaluate the deferred computation; (2) we introduce a dummy phantom node to anchor objects pointed to by formal parameters and return values that are object references; (3) we introduce phantom nodes for handling upward exposed objects; (4) for every access path in the concrete storage graph, we can compute a corresponding path in the connection graph; and (5) the use of phantom nodes allows us to analyze a method independent of the calling context.

3. INTRAPROCEDURAL ANALYSIS

Given the control flow graph (CFG) representation of a Java method, we use a simple iterative scheme for constructing the intraprocedural connection graph. We describe two variants of our analysis, a flow-sensitive version, and a flow-insensitive version. To simplify the presentation, we assume that all multiple-level reference expressions of the form `a.b.c.d...` are split into a sequence of simple two level reference expressions that are of the form `a.b`. Any bytecode generator automatically does this simplification for us. For example, a Java statement of the form `a.b.c.d = new T()` will be transformed into a sequence of simpler statements: `t = new T(); t1 = a.b; t2 = t1.c; t2.d = t;` where `t, t1,` and `t2` are new temporary reference variables of the appropriate type.

Given a node s in the CFG, the connection graph at entry to s (denoted as C_i^s) and the connection graph at exit from s (denoted as C_o^s) are related by the standard data flow equations:

$$C_o^s = f^s(C_i^s) \quad (1)$$

$$C_i^s = \wedge_{r \in Pred(s)} C_o^r, \quad (2)$$

where f^s denotes the *data flow transfer function* of node s. The *meet* operation ($\wedge$) is a merge of connection graphs. We define a merge between two connection graphs $C_1 = (N_1, E_1)$ and $C_2 = (N_2, E_2)$ to be the union of the two graphs. Furthermore, if N_1 and N_2 have common nodes, i.e. nodes with the same unique node id, the escape state of the corresponding node in the merged graph is a meet of the common nodes' escape states. More formally,

$$C_3 = C_1 \wedge C_2 = (N_1 \cup N_2, E_1 \cup E_2)$$

and let n_3 a node in C_3. The escape state of $n_3 \in C_3$ is:

$$n_3.es = \begin{cases} n_1.es \wedge n_2.es & \text{if } \exists n_1 \in N_1 | n_1.id = n_3.id, \exists n_2 \in N_2 | n_2.id = n_3.id, \\ n_1.es & \text{if } \exists n_1 \in N_1 | n_1.id = n_3.id, \nexists n_2 \in N_2 | n_2.id = n_3.id, \\ n_2.es & \text{if } \nexists n_1 \in N_1 | n_1.id = n_3.id, \exists n_2 \in N_2 | n_2.id = n_3.id \end{cases} \quad (3)$$

Even though for simplicity, we talk about different connection graphs at different program points, in our implementation, we maintain a single connection graph for each method which is updated incrementally. We handle loops by iterating over the data flow solution until it converges. We impose an upper limit (our current implementation uses an upper bound of ten) on the number of iterations – if convergence is not reached, a `bottom` solution, in which every node is marked *GlobalEscape*, is assumed for that method.

Given the bytecode simplification of Java programs, we identify four *basic statements* that affect intraprocedural escape analysis: (1) p = new $\tau()$, (2) $p = q$, (3) $p.f = q$, (4) $p = q.f$. We present the transfer functions for each of these statements. Figure 5 illustrates the transfer functions for each of the basic statements for flow-sensitive analysis. Note that all the variables on the left-hand-side of the statements are local variables: global variables (i.e. static fields) are never killed.

Figure 6 illustrates an example showing the connection graphs at various program points computed using the analysis described in this section.[4] The first column of the figure corresponds to C_i^s, the middle column corresponds to the statement s, and the third column corresponds to C_o^s.

(1) $p = new\ \tau()$. We first create a new object node O (if one does not already exist for this site). For flow-sensitive analysis, we first apply $ByPass(p)$ and then add a new points-to edge from p to O. For flow-insensitive analysis, we do not apply $ByPass(p)$, but simply add the points-to edge from p to O.

(2) $p = q$. As in the previous case, for flow-sensitive analysis, we first apply $ByPass(p)$, and then add the edge $p \xrightarrow{D} q$. Again, for flow-insensitive analysis we ignore $ByPass(p)$ but add the edge $p \xrightarrow{D} q$. The difference is that we can *kill* what p points to with flow-sensitive analysis, but not with flow-insensitive analysis.

(3) $p.f = q$. Let $U = PointsTo(p)$. If $U = \emptyset$, then either (1) p is null (in which case, a null pointer exception will be thrown), or (2) the object that p points to was created outside of this method (this could happen if p is a formal parameter or reachable from a formal parameter). We conservatively assume the second possibility (if $U = \emptyset$) and create a *phantom object node* O_{ph}, and insert a points-to edge from p to O_{ph} (if p is null, the edge from p to O_{ph} is spurious, but does not affect the correctness of our analysis).
During interprocedural analysis, the phantom nodes will be mapped back to the actual nodes (see Definition 4.1 of the *MapsTo* function) created by the appropriate procedure. We also use a 1-limited scheme for creating phantom nodes. Now let $V = \{v | u \xrightarrow{F} v$ **and** $u \in U$ **and** $fid(v) = f\}$, where $fid(v)$ is the

[4]In order to keep the figure simple, we have not transformed a statement like `a.f = new T1()` to its equivalent form: `t = new T1(); a.f = t;`.

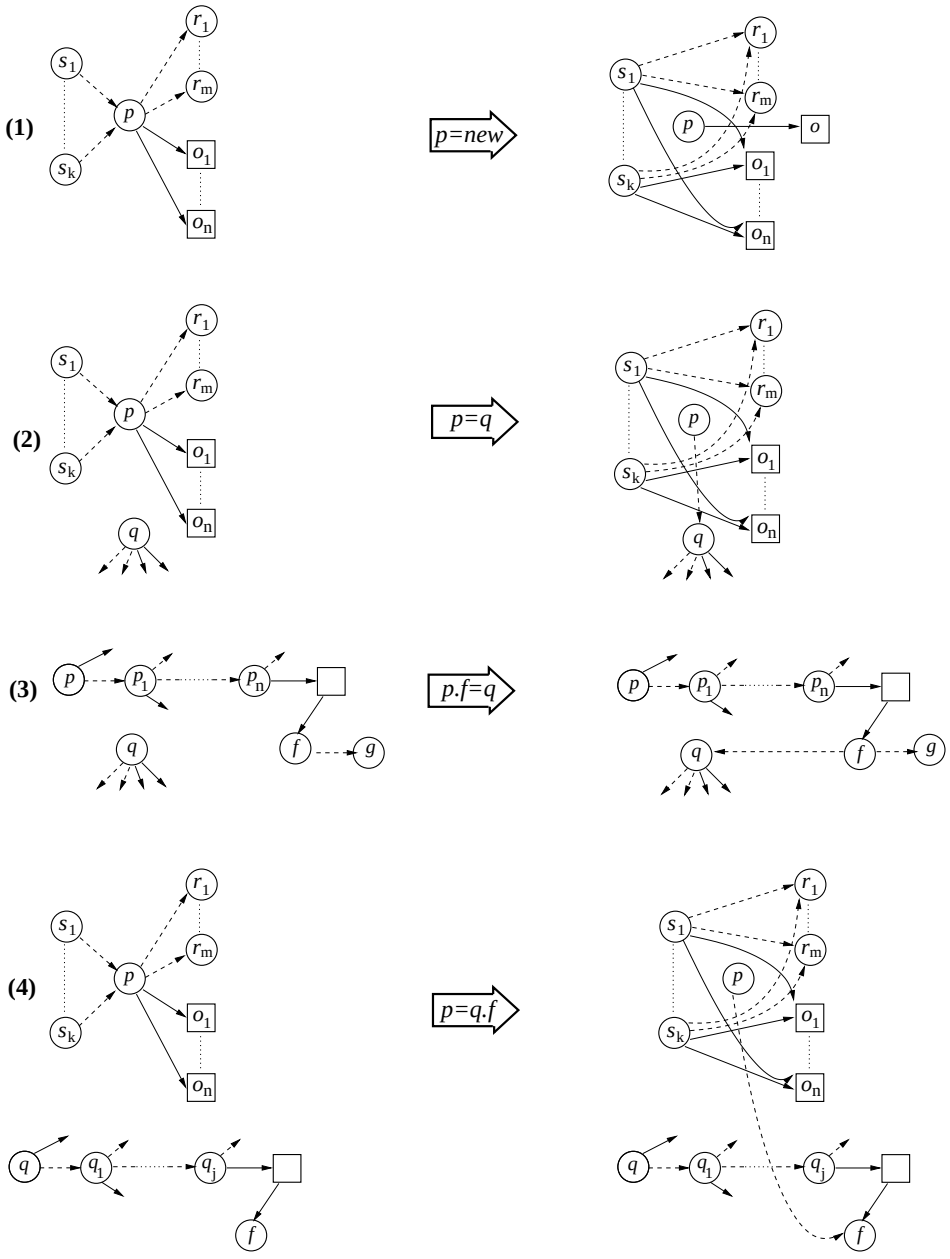


Fig. 5. Flow-sensitive transfer functions for basic statements.

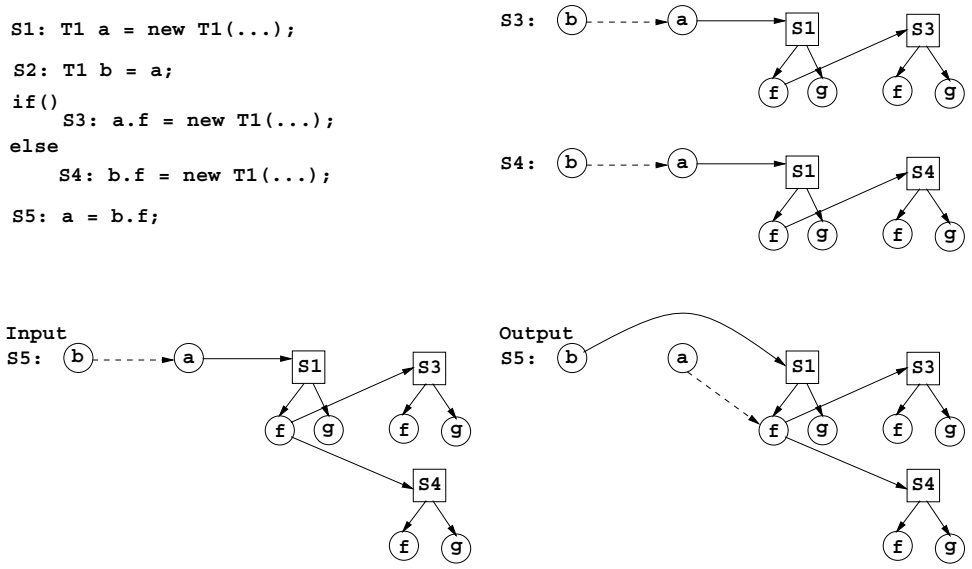


Fig. 6. An example illustrating connection graph computation. The connection graphs at S1 and S2 are not shown.

field id of the field node v. Again, it is possible that V is empty. In this case, we create a field reference node (lazily) and add it to V. Finally we add edges in $\{v \xrightarrow{D} q | v \in V\}$ to the connection graph. Note that even for flow-sensitive analysis, we cannot in general kill whatever $p.f$ was pointing to, and so we do not apply $ByPass(p.f)$. This is because even a single object that p points to in any k-limited representation may correspond to more than one concrete object.

(4) $p = q.f$. Let $U = \{u | q \xrightarrow{+1} u\}$, $V = \{v | u \xrightarrow{F} v$ **and** $u \in U$ **and** $fid(v) = f\}$. As in the previous case, if U is empty, we create a phantom node and add it to U, and if V is empty, we create a field reference node and add it to V.
For flow-sensitive analysis, we first apply $ByPass(p)$, and then add the edges in $\{p \xrightarrow{D} v | v \in V\}$ to the connection graph. For flow-insensitive analysis we once again ignore $ByPass(p)$, but add the edges in $\{p \xrightarrow{D} v | v \in V\}$ to the connection graph.

4. INTERPROCEDURAL ANALYSIS

The core part of our analysis proceeds in a bottom-up manner, in which the summary information (in the form of a subgraph of CG) obtained for a callee is used to update the CG of the caller. A key contribution of our analysis is the manner in which we summarize the effect of a method, so that a single succinct summary can be accurately used for different calling contexts while determining the stack-allocatability of objects. Note that for pointer analysis, which is closely related to our analysis but solves a more general problem, a procedure in general cannot be accurately summarized independent of the aliasing relationships that hold at

its caller [Landi and Ryder 1992; Choi et al. 1993; Emami et al. 1994; Wilson and Lam 1995; Ghiya and Hendren 1998; Chatterjee et al. 1999]. (An exception is a type-based alias analysis, which ignores execution flow in a program.) In the absence of cycles in the program call graph (PCG), a single traversal of nodes in reverse topological order over the PCG would be sufficient for this phase. In order to handle cycles (due to recursion), we iterate over the nodes in strongly connected components of PCG in a bottom-up manner until the data flow solution converges. As with intraprocedural analysis, we impose a constant upper bound on the number of iterations, and assume a `bottom` solution if convergence is not reached within those iterations – i.e., all nodes in the set N_a (for actual arguments and return value) for methods involved in a non-converging strongly connected component are marked *GlobalEscape*. This phase is sufficient to identify stack-allocatable objects (which are also thread-local).

In order to identify additional thread-local objects, i.e., those which are not stack-allocatable, we need to propagate some information from the caller to its callees. This step, which constitutes an extension to our core analysis, is performed in a separate top-down pass over the PCG.

The remainder of this section is organized as follows. Sections 4.1 through 4.4 describe the core interprocedural analysis at four points of interest: (1) method entry, (2) method exit, (3) immediately before a method invocation, and (4) immediately after a method invocation. Section 4.5 describes the extension to identify more thread-local objects. Section 4.6 discusses how Java's type-safety could be exploited when dealing with conservative situations such as native method calls.

We will use the Java example shown in Figure 7 to illustrate our interprocedural framework. In this example, method $L()$ constructs a linked list and method $T()$ constructs a tree-like structure. Figure 7(B) shows the caller-callee relation for the example program, shown in Figure 7(A). In Figure 7(B), we identify the four points of interest to interprocedural analysis, which are discussed in the following subsections. Figure 8 shows the connection graphs built at various points of the program in Figure 7(A). These connection graphs are a conservative representation of the data structure built by the program during execution.

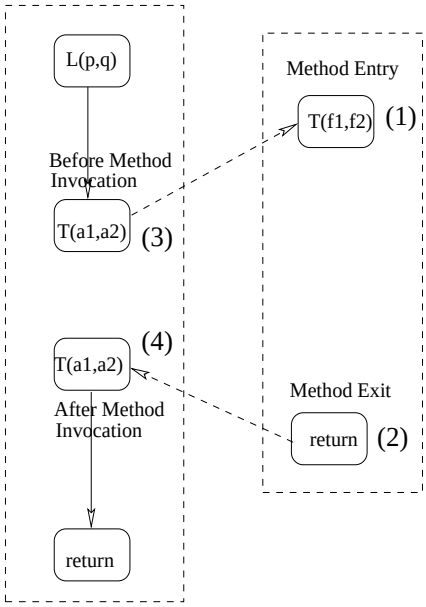
4.1 Connection Graph at Method Entry

We process each formal parameter (of reference-type) in a method one at a time. Note that the implicit `this` reference parameter for an instance method appears as the first parameter. For each formal parameter f_i, there exists an actual parameter a_i in the caller of the method that produced the value for f_i. Nodes corresponding to actuals belong to N_a. At the method entry point, we can envision an assignment of the form $f_i = a_i$ that copies the value of a_i to f_i. Since Java specifies call by *value* semantics, f_i is treated like a local variable within the method body, and so it can be killed by other assignments to f_i. We create a *phantom reference node* for a_i and insert a deferred edge from f_i to a_i. The phantom reference node serves as an anchor for the summary information that will be generated when we finish analyzing the current method.[5] We initialize $EscapeState[f_i] = NoEscape$ and

[5]We use a_i as the anchor point rather than f_i, since, in Java, f_i is treated as a local variable, and so the deferred edge from f_i to a_i can be deleted.

```
class ListElement {

   int data;
   ListElement next;
   static ListElement g = null;
   ListElement() {data = 0; next = null;}

   static void L(int p, int q) {
S0:   ListElement u = new ListElement();
      ListElement t = u;
      while(p > q) {
S1:     t.next = new ListElement();
        t.data = q++;
        t = t.next;
      }
S2:   ListElement v = new ListElement();
      NewListElement.T(u, v);
   }
}

class NewListElement {

   ListElement org;
   NewListElement next;
   NewListElement() {org = null; next = null;}

   static ListElement T(ListElement f1, ListElement f2) {
S3:   NewListElement r = new NewListElement();
      while(f1 != null) {
S4:     r.org = f1.next;
S5:     r.next = new NewListElement();
        . . . // do some computation using r
        . . . // w/o changing the data structure
S6:     r = r.next;
        if(f1.data == 0)
S7:       ListElement.g = f2;
        f1 = f1.next;
      }
      return f2;
   }
}
```

(A)



(B)

Fig. 7. An example program for illustrating interprocedural analysis and its call graph.

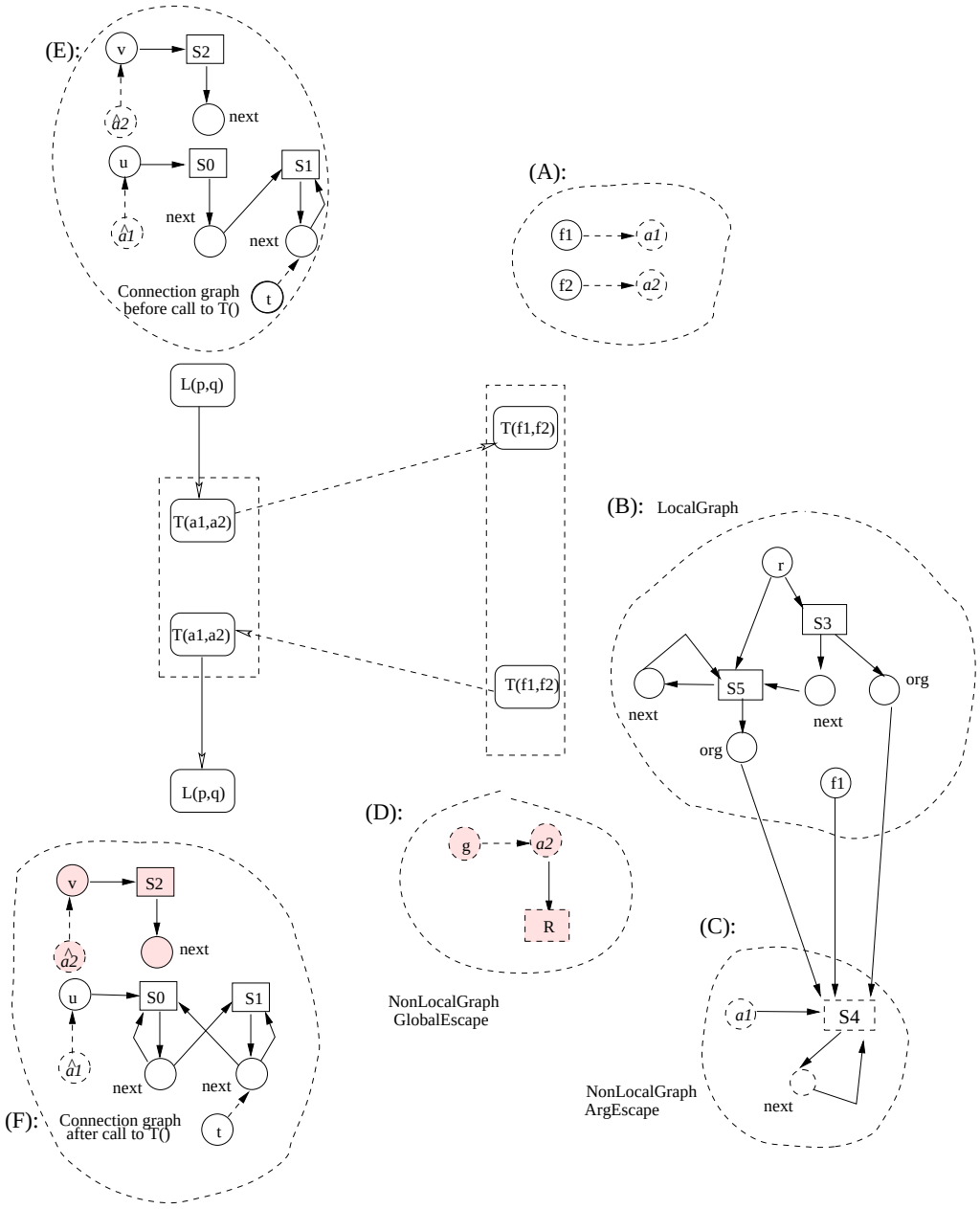


Fig. 8. Connection graphs at various points in the call graph. Nodes that escape globally are shadowed.

$EscapeState[a_i] = ArgEscape$. Figure 8(A) illustrates the reference nodes `f1` and `f2`, the phantom reference nodes *a1* and *a2*, and the corresponding deferred edges at the entry of method `T()`.

4.2 Connection Graph at Method Exit

We model a `return` statement that returns a reference to an object as an assignment to a special phantom variable called *return* (similar to formal parameters). Multiple return statements are handled by "merging" their respective return values. We model each `throw` statement conservatively by marking the object being thrown as *GlobalEscape*. After completing intraprocedural escape analysis for a method, we use the *ByPass* function (defined in Section 3) to eliminate all the deferred edges in the CG, creating phantom nodes wherever necessary. For example, Figure 8(B) - Figure 8(D) show the connection graph at the exit of method `T()`, and the phantom node R in Figure 8(D), representing the return node, is created during this process. Handling return values due to an exception are described in Section: 5.

We then do reachability analysis on the CG holding at the return statement of the method to update the escape state of objects. The reachability analysis partitions the graph into three subgraphs:

(1) The subgraph induced by the set of nodes that are reachable from a *GlobalEscape* node. The initial nodes marked *GlobalEscape* are static fields of a class and `Runnable` objects. This subgraph is collapsed into a single *bottom* node that efficiently represents all the nodes whose escape state is *GlobalEscape*.
(2) The subgraph induced by the set of nodes that are reachable from an *ArgEscape* node, but not reachable from any *GlobalEscape* node. The initial *ArgEscape* nodes are the phantom reference nodes that represent the actual arguments created at the entry of a method, such as *a1* and *a2* in Figure 8(A), and the phantom node for the *return* variable.
(3) The subgraph induced by the set of nodes that are not reachable from any *GlobalEscape* or *ArgEscape* node (which remain marked *NoEscape*).

We call the union of the first and the second subgraphs the *non-local subgraph* of the method, and the third subgraph the *local subgraph*. It is easy to show that there can only be edges from the local subgraph to the non-local subgraph, but not *vice versa*. All objects in the local subgraph that are created in the current method are marked stack-allocatable. The non-local subgraph represents the summary connection graph of the method. This summary information is used at each call site invoking the method, as described in the next section. As a further optimization to reduce the number of nodes and path lengths in the summary representation, each reference node in the non-local subgraph is bypassed by connecting its predecessors directly to its successors, so that the non-local subgraph consists only of the nodes representing actual parameters, objects and fields accessed via the parameters, and a single bottom node that represents all nodes with the *GlobalEscape* escape state.

Figure 8(B) - Figure 8(D) show the connection graph at the exit of method `T()`. In this connection graph, the object node S4 is a phantom node that was created at Statement S4 during intraprocedural analysis of `T()`. The object nodes S3 and S5 were created locally in `T()`. In the figure, we can see that the structure in Figure 8(B) is local to method `T()`, and so will not escape `T()`. We also see that the

assignment to the global reference variable, "`g = f2`", makes the formal parameter `f2` and the phantom reference node *a2* all *GlobalEscape* as shown in Figure 8(D). In the figure, a deferred edge from `g` to *a2* is shown for exposition. The summary graph for method `T()` will consist of the non-local subgraphs shown in Figure 8(C) and Figure 8(D). This summary graph will be mapped back to caller's connection graph (see Section 4.4).

4.3 Connection Graph Immediately Before a Method Invocation

At a method invocation site, each parameter passing is handled as an assignment to an actual parameter $\hat{a_i}$ at the caller. Let u_1 be a reference to an object U_1. Consider a call $u_1.foo(u_2, \ldots u_n)$, where $u_2 \ldots u_n$ are actual parameters to $foo()$. We model the call as follows: $\hat{a_1} = u_1; \hat{a_2} = u_2; \ldots ; foo(\hat{a_1}, \hat{a_2}, \ldots \hat{a_n})$. How to handle a virtual method is described in the next section. Each $\hat{a_i}$ at the call site will be matched with the phantom reference node a_i of the callee method. In Figure 8(E), two nodes, $\hat{a1}$ and $\hat{a2}$, are created with deferred edges pointing to the first and the second actual parameters to the call, `u` and `v`, respectively.

4.4 Connection Graph Immediately After a Method Invocation

At this point, we essentially map the callee's connection graph summary information back to the caller's connection graph (CG). Three types of nodes play an important role in updating the caller's CG with the callee's CG immediately after a method invocation: $\hat{a_i}$'s of the caller's CG, a_i's of the callee's CG, and the `return` node of the callee's CG. Updating the caller's CG is done in two steps: (1) updating the node set of the caller's CG using $\hat{a_i}$'s and a_i's; and (2) updating the edge set of the caller's CG using $\hat{a_i}$'s and a_i's. We refer to these steps collectively as the `UpdateCaller()` routine. Updating the `return` node is done during the first step by treating the `return` node the same as a_i and treating the node that is assigned the result of the method invocation the same as $\hat{a_i}$. If the callee is a virtual method, we first union the summary connection graphs of all the methods that might be a target of the call, and then use the unioned graph as the callee's summary connection graph.

Updating Caller Nodes. Figure 9 describes how we map the nodes in the callee's CG with the nodes in the caller's CG. This mapping of nodes from callee CG to caller CG is based on identifying the *MapsTo* relation among object nodes in the two CGs. As a base case, we ensure that a_i maps to $\hat{a_i}$. Given the base case, we also ensure that a node in $PointsTo(a_i)$ maps to any node in $PointsTo(\hat{a_i})$. We formally define the relation *MapsTo* ($\longmapsto$), among objects belonging to a callee CG and a caller CG recursively as follows:

Definition 4.1. (1) $a_i \longmapsto \hat{a_i}$

(2) $O_p \in PointsTo(p) \longmapsto O_q \in PointsTo(q)$, if
  (a) $(p = a_i) \wedge (q = \hat{a_i})$, or
  (b) $(p = O.f) \wedge (q = \hat{O}.g) \wedge (O \longmapsto \hat{O}) \wedge (fid(f) = fid(g))$.

In Figure 9, $MapsToObj(n)$ denotes the set of objects that n can be mapped to using the *MapsTo* relation discussed above. In the figure, we use the subscript *er* to denote caller nodes and *ee* to denote callee nodes. The algorithm starts with a_i and

```
UpdateCallerNodes()
{
1:    foreach a_i, â_i actual parameter pair do
2:      UpdateNodes (a_i, {â_i});
3:    endfor
}
UpdateNodes( f_ee: field node;
             MapsToF: set of field nodes)
// MapsToF is the set of MapsTo field nodes of f_ee
{
4:    foreach object node n_o ∈ PointsTo(f_ee) do
5:      foreach f_er ∈ MapsToF do
6:        if PointsTo(f_er) = ∅ then
7:          CreateTargetNode( f_er ); // create/insert a new node as the target of f_er
8:        endif
9:        foreach n̂_o ∈ PointsTo(f_er) do
10:         if n̂_o ∉ MapsToObj(n_o) then
11:           MapsToObj(n_o) = MapsToObj(n_o) ∪ {n̂_o};
12:           UpdateEscapeState(n_o, n̂_o);
13:           foreach f´_ee such that  n_o --F--> f´_ee do
14:             tmpMapsToF = {f´_er | n̂_o --F--> f´_er and fid(f´_ee) = fid(f´_er)};
15:             UpdateNodes( f´_ee, tmpMapsToF);
16:           endfor
17:         endif
18:       endfor
19:     endfor
20:   endfor
}
```

Fig. 9. Algorithm to Update the Caller's Connection Graph Nodes.

$\hat{a_i}$ as the original "fields" that map to/from each other, and then recursively finds other objects in the caller CG that are *MapsTo* nodes of each corresponding callee object. The escape state of the nodes in $MapsToObj(n)$ is marked *GlobalEscape* if the escape state of n is *GlobalEscape* (`UpdateEscapeState()` at Statement 12). Otherwise, the escape state of the caller nodes is not affected.

The main body of procedure `UpdateNodes` is applied to all the callee object nodes pointed to by the callee field node f_{ee} (Statement 4). Given a callee object node n_o, Statement 9 computes the set of n_o's *MapsTo* object nodes in the caller graph. This is done by identifying the set of caller object nodes "pointed" to by the caller field node f_{er}, which is itself a *MapsTo* field node of callee node f_{ee} (i.e. $f_{er} \in MapsToF$). A caller object node, $\hat{n_o}$, and its field nodes are created at Statement 7 if no *MapsTo* caller object node exists, with an escape state of *NoEscape* (`CreateTargetNode()` at Statement 7). In the connection graph of a method, however, we create at most one object node for any allocation site in the program.

Given a callee object node n_o and its *MapsTo* caller node $\hat{n_o}$, Statement 14 computes, for each field node of n_o (i.e. $f\acute{}_{ee}$), the set of *MapsTo* field nodes of the caller (i.e. *tmpMapsToF*). It then recursively invokes `UpdateNodes`, passing $f\acute{}_{ee}$ and *tmpMapsToF* as the new parameters (Statement 15).

Updating Caller Edges. Recall that following the removal of deferred edges, there are two types of edges in the summary connection graph: field edges and points-to edges. Field edges get created at Statement 7 in Figure 9 while the nodes are updated.

To handle points-to edges, we do the following: Let p and q be object nodes of the callee graph such that $p \xrightarrow{F} f_p \xrightarrow{P} q$. Then, for each $\hat{p} \in MapsToObj(p)$ and $\hat{q} \in MapsToObj(q)$, both of the caller, we establish $\hat{p} \xrightarrow{F} \hat{f}_p \xrightarrow{P} \hat{q}$ by inserting a points-to edge $\hat{f}_p \xrightarrow{P} \hat{q}$ for each field node $\hat{f}_p$ of $\hat{p}$ such that $fid(f_p) = fid(\hat{f}_p)$.

Example. Consider the summary non-local subgraphs shown in Figure 8(C) and Figure 8(D). First, all nodes that are reachable from global variable `g` are marked $\perp$ (i.e. *GlobalEscape*). Then, all nodes reachable from the phantom reference node *a1*, but not reachable from `g` are marked as *ArgEscape*. Now when we analyze method `L()` intraprocedurally we would construct the connection shown in Figure 8(F) that is right after the invocation site of `T()`. We will first mark the phantom reference node *a1* of the callee (in Figure 8(C)) and the phantom node $\hat{a1}$ of the caller (in Figure 8(F)) as the initial "field" nodes, i.e. a_i and $\{\hat{a}_i\}$ at Statement 2 in Figure 9. Then we will map the phantom node `S4`, pointed to by *a1*, to `S0`, pointed to by $\hat{a1}$. The cycle in the non-local subgraph of `T()` also results in mapping `S1` as a *MapsTo* node of `S4`. The cycle also results in inserting edges from the `next` fields of `S0` and `S1` to both `S0` and `S1`. This is a result of the 1-limited approach we take in creating a phantom node: we create at most one phantom node at a statement. Now since *a2* is marked $\perp$, all the nodes of the caller reachable from $\hat{a2}$ will also be marked as $\perp$.

4.5 Updating Escape State of Nodes in Callees to Identify Thread-Local Data

After the bottom-up phase of interprocedural analysis described above, all object nodes marked *NoEscape* can be regarded as stack-allocatable as well as thread-local. We now describe the analysis step to identify those object nodes, marked *ArgEscape* at this stage, which should be marked *GlobalEscape* based on reachability information from any caller. If this step is omitted, all object nodes marked *ArgEscape* will have to be conservatively regarded as escaping their thread of creation. For simplicity, we perform this propagation of *GlobalEscape* state in a separate top-down traversal over the PCG, although it could be combined with the core interprocedural analysis described above. Again, cycles in the PCG are handled by iterating until the solution converges. For each object marked *GlobalEscape* in a method, we identify the corresponding nodes in each callee method and mark them *GlobalEscape*. Thus, an object in a method is conservatively marked as escaping its thread of creation if it escapes the thread of creation in any caller of that method. (This conservativeness is necessary because we do not perform method specialization.) The nodes in the callees which correspond to nodes in the callers are identified using the inverse of the *MapsTo* relation described in Section 4.4. In fact, we keep track of this inverse mapping when applying the analysis described in Section 4.4. Note that this step does not affect the escape state of *any* node marked *NoEscape* in the core part of our interprocedural analysis, because if such a node had a corresponding node in the caller (i.e., if it were reachable from the caller), it would not have been

marked *NoEscape* in the first place.

4.6 Java's Strong Type Information

We can exploit Java's strong type system in computing the connection graph for a method whose body cannot be (or, has not been) analyzed. Examples of bottom methods are native methods implemented in a non-Java language. The representation for such a method, called a *bottom method*, is called the *bottom graph*, which has one node for each class of the program. Given two nodes N_1 and N_2 in the bottom graph that represent two classes C_1 and C_2, respectively, there is a points-to edge from N_1 to N_2 if C_1 contains an instance field that is a reference to C_2. There is a deferred edge from N_1 to N_2 if C_2 is a sub-type of C_1. If a static field of any class contains a reference to C_1, the corresponding node N_1 in the bottom graph is marked *GlobalEscape*.

In effect, the bottom graph is the most conservative connection graph of the program allowed under Java's type system.

The bottom graph can be used to conservatively establish connections among nodes that are reachable from the actual parameters passed to a bottom method and static fields. Let O_1 and O_2 be two object nodes reachable from actual parameters passed to a bottom method or from static fields, and N_1 and N_2 be their corresponding nodes in the bottom graph. If there exists an edge from a field of N_1 to N_2, we insert a edge from the corresponding field node of O_1 to O_2. Also, we make *GlobalEscape* any object reachable from an actual parameter passed to a bottom method if the corresponding node in the bottom graph is marked *GlobalEscape*.

In a dynamic optimization system, a method that has not been analyzed by the compiler also becomes a *bottom method* when the compiler generates code for a caller of the method. In this case, the bottom method may have been interpreted or compiled without analysis/optimization. Although our current implementation does not take advantage of the type information in bottom methods, the combination of the bottom graph and the summary graph would make our approach for escape analysis well suited for dynamic Java compilation systems such as Jalapeño at IBM Research [Burke et al. 1999].

5. HANDLING EXCEPTIONS AND FINALIZATION OF JAVA

In this section, we show how we handle Java-specific features such as exceptions and object finalization.

5.1 Exceptions

We now show how our framework handles exceptions. Exceptions are *precise* in Java, hence any code motion across the exception point should be invisible to the user program. An exception thrown by a statement is caught by the closest dynamically enclosing `catch` block that handles the exception [Gosling et al. 1996].

One way to do data flow analysis in the presence of exceptions is to add a control flow graph edge from each statement that can throw an exception to each `catch` block that can potentially catch the exception, or to the exit of the method if there is no `catch` block for the exception. The added edges ensure that data flow information holding at an exception-throwing statement will not be killed by state-

ments after the exception throwing statement, since the information incorporating the "kill" would be incorrect if the exception was thrown.

We, however, use a simpler strategy for doing data flow analysis in the presence of exceptions. Recall that we "kill" only local reference variables of a method in a flow-sensitive analysis. Therefore, we only need to worry about them. Of the local variables within a `try` block, we kill only those that are declared within the block. Local reference variables declared outside the `try` block should not be killed, as they can be live at the termination of the block if an exception is thrown. We will use the following example to elaborate on this point. In the example, `x` is local to the method, but non-local to the `try-catch` statement.

```
m0( T1 f1, T2 f2) {
     T1 x;
S1: try {
S2:  x = new T1(); // creates object O1
S3:  x.b = f2;
         // sets up a path from x to f2.
S4:  ... // an exception is thrown here.
S5:  x = new T1(); // creates object O2
    } catch (Exception e) {
S6:   System.out.println("Don't worry");
    }
S7: f1.a = x;
}
```

Assume that an exception is thrown at $S4$. After the `catch` block, when $S7$ is executed, `f2` will become reachable from `f1`. If we were to kill the points-to edge from `x` to object node O1 at $S5$, then we would lose the path information from `f1` to `f2`, and hence, would have an incorrect connection graph. Recall that our strategy is not to kill information for variables in a `try` block that are not local to the block. Hence, in this example, we will not delete the previous edge from `x` to `O1` (whose field node `b` has an edge to `f2`) while analyzing $S5$. Hence, at $S7$, after putting an edge from `f1` to `x`, we would correctly have a connection graph path from `f1` to `f2`.

A method (transitively) invoked within a `try-catch` block can be handled in the same manner as a regular statement block in its place: we can kill any locals declared in that method. An important implication of this approach is that we can ignore potential run-time exceptions within methods that do not have any `try-catch` blocks in them. Many methods in Java correspond to this case.

5.2 Finalization

Before the storage for an object is reclaimed by the garbage collector, the Java Virtual Machine invokes a special method, the *finalizer*, of that object [Gosling et al. 1996]. The class `Object`, which is a superclass of every other class, provides a default definition of the `finalize` method, which takes no action. If a class overrides the `finalize` method such that its `this` parameter is referenced, it means

that an object of that class is reachable (due to the invocation of the finalizer) even after there are no more references to it from any live thread. We deal with this problem by marking each object of the class overriding the finalizer as *GlobalEscape* ($\perp$).

6. ELIMINATING REDUNDANT STORAGE SYNCHRONIZATION OPERATIONS

Synchronization operations in Java perform two functions. The first is to enforce mutual exclusion by acquiring and releasing a lock. These locking operations can be optimized as discussed in Section 7. The second function is to force global values cached in thread local memory to be updated from, or written to, global memory. This is similar to the effects of synchronization in a *release consistency* system such as TreadMarks [Amza et al. 1996].

The semantics of the *acquire* phase of the synchronization operation are to (1) obtain a lock, and (2) ensure that locally cached values of global shared variables are updated with the global shared value. The second requirement can be enforced by the hardware cache coherence mechanism and an inexpensive instruction synchronization (*isync* on the PowerPC architecture) instruction. The semantics of the *release* phase of the synchronization operation are to (1) make sure all writes to memory of global shared values are completed, and (2) release the lock. Enforcement of the first requirement requires executing a storage synchronization instruction. In this article the storage synchronization instruction will be called the *sync* instruction, which is its name in the PowerPC architecture [May et al. 1994][6] *Sync* operations can be expensive, particularly on multiprocessor systems, since each memory sub-system must acknowledge the completion of all writes. The remainder of this section discusses an unimplemented analysis that builds upon the results of escape analysis to determine when *sync* operations can be eliminated from the release phase of a synchronization operation.

The results in Table III of Section 7 give an upper bound of the effectiveness of this analysis on a uniprocessor system. Because the cost of a *sync* operation is typically higher on a multiprocessor system, the benefit from this analysis will also be greater.

6.1 When *sync* operations are required

Before considering the details of how to determine when a *sync* operation is not needed, we state the conditions when a *sync* operation must be executed. Let w (r) be a write (read) of value v in thread T_w (T_r) of a thread escaping object O. Also let s be a *sync* operation implied by the release phase of some synchronization in T_w. Then s must be executed if (1) there is a path from any w to s without an intervening *sync* operation, and (2) there exists a corresponding r operation for the w operation.

In our analysis we make the following conservative assumption: for any write w and r as defined above, we assume that if w exists then r also exists, and $T_r \neq T_w$. Because escape analysis information has already been computed, determining that a *sync* operation s should be eliminated only requires determining if there is a path

[6]The *sync* instruction is sometimes called the *fence* instruction in the literature and on other architectures.

from some w to s that does not contain another *sync* operation. We model this as a 1-bit dataflow problem.

6.2 Intraprocedural analysis

The intraprocedural analysis is accomplished by iterating over a control flow graph of the method or procedure being analyzed. The graph contains nodes representing acquire operations, branches, and writes to global escaping objects. Each node N of the graph has four fields: *IN*, *OUT*, *GEN* and *KILL*. The *GEN* field is initially set to **true** if the node contains a write to a thread escaping object, and **false** otherwise. The *KILL* node is set to **false** if the node performs a release operation, and **true** otherwise. The *IN* and *OUT* nodes are initialized to **false**. If the node represents a method invocation, the values of the fields represent summary information for the method as described in Section 6.3.

Nodes other than those representing acquire operations, and writes to global escaping objects have no affect on the analysis, and can be modeled by setting their *OUT* field to the value of their *IN* field whenever they are visited.

Intuitively, the *IN* field of a node n_i becomes **true** after the analysis if there is a path from a write of a thread escaping variable to n_i which does not visit a node containing a release operation, and remains **false** otherwise. *OUT* becomes true if either this node contains a write of a thread escaping variable, or *IN* is **true** and the node does not perform a release operation.

The entry node for the method is initialized differently, with *GEN* set to **true**. This allows the optimizations to be correct regardless of the calling context.

The analysis iterates over a method's graph until no change is noted in the *KILL* fields for all nodes. As each node N is visited, the *IN* field is computed as:

$$IN = \bigvee_{p \in pred(N)} p.IN \tag{4}$$

and the transfer function is computed by:

$$OUT = (IN \wedge KILL) \vee GEN. \tag{5}$$

The $\wedge$ operation is performed using a bit-wise *and*, while $\vee$ is performed using a bit-wise *or* operation. Intuitively, if *KILL* is true (i.e. the node does not perform a release) it does not effect the state of the input to the node. After the analysis is completed, any node that performs a release operation, and whose *IN* field is **true** must perform a *sync* as part of the release operation. The *sync* can be eliminated from all other release operations.

6.3 Procedure summary information

Procedure summary information is gathered by invoking the intra-procedural analysis algorithm at each call to a method for which summary information is not available. Summary information is gathered for the *GEN* and *KILL* fields. If the invocation is a virtual method call, then the summary *GEN* information for the call site is computed by *or*'ing the *GEN* information for all methods that might be invoked. The summary *KILL* information is found by performing a *meet* operation ($\wedge$) over the *KILL* information for all methods that might be called.

GEN summary information is gathered by setting the *IN* field of the ENTRY node of the graph for the method being analyzed to **false**. If, after analysis, the *OUT* field of the graph's *exit* node is **false**, then either no writes to thread escaping objects occurred in the method, or any that did were killed by release operations. The summary *GEN* information is also set to **false**. Otherwise, the summary information is set to **true**.

To compute *KILL* summary information, the *IN* field of the *entry* node of the graph for the method being analyzed is set to **true**. If, after analysis, the *OUT* field of the graph's *exit* node is **false**, then both the incoming write to a thread escaping object field is killed, or *sync*'ed, within the thread, and no additional write to a thread escaping object's field is performed that is not also killed. Thus the invocation as a whole kills the incoming writes, and the summary *KILL* information is set to **false**. Otherwise, it is set to **true**. Note that if the *GEN* summary information for a method is **true**, the *KILL* information must also be **true**.

The computed summary information is memoized, and therefore only needs to be computed once for each method, not once per call site. *GEN* and *KILL* summary information can be conservatively approximated by setting them to both to **true**.

7. TRANSFORMATION AND RUN-TIME SUPPORT

We have implemented two optimizations based on escape analysis in the IBM High Performance (static) Compiler for Java (HPCJ) for the PowerPC/AIX architecture platform [International Business Machines Corporation 1997; Seshadri 1997]: (1) allocation of objects on the stack, and (2) elimination of unnecessary synchronization operations. In this section, we describe the transformation applied to the user code (based on the analysis described in previous sections) and the run-time support to implement these optimizations.

7.1 Transformation

Once the analysis converges during the iteration over the call graph (i.e., when there are no further changes being made to any connection graph in terms of edges or the *EscapeState* of nodes), we mark each `new` site in the program as follows, based on the following information: (1) if the *EscapeState* of the corresponding object node is *NoEscape*, the `new` site is marked stack-allocatable, and (2) if the *EscapeState* of the corresponding object node is *NoEscape* or *ArgEscape*, the `new` site is marked as allocating thread-local data. Since we use a 1-limited scheme for naming objects, a `new` statement (a compile-time object name) is marked stack-allocatable or thread-local only if all objects allocated during run time at this `new` site are stack-allocatable or thread-local, respectively.

7.2 Run-Time Support

We allocate objects on the stack by calling the native `alloca` routine in HPCJ's AIX backend. Each invocation of `alloca` essentially grows the current stack frame at run time by some amount. In cases where (1) the object requires a fixed size, and (2) either just a single instance of a `new` statement executes in a given method invocation, or the previous instance of the object allocated at a `new` statement is no longer live when the `new` statement is executed next, it would be possible to allocate a fixed piece of storage on the stack frame for that `new` statement. Our current

Program	Description	Number of classes	Size of classes
vtrans	High Performance Java Translator (IBM)	142	503K
jgl	Java Generic Library 1.0 (ObjectSpace)	135	217K
jacorb	Java Object Request Broker 0.5 (U. Freie)	436	308K
jolt	Java to C translator (KB Sriram)	46	90K
jobe	Java Obfuscator 1.0 (E. Jokipii)	46	60K
javacup	Java Constructor of Parsers (S. Hudson)	59	101K
hashjava	Java Obfuscator (KB Sriram)	98	183K
toba	Java to C translator (U. Arizona)	19	86K
wingdis	Java decompiler, demo version (WingSoft)	48	178K
pbob	portable Business Object Benchmark (IBM)	65	333K

Table I. Benchmarks used in our experiments.

implementation does not perform this analysis to reuse stack space. A potential downside of our approach is that the stack frame may grow in an unconstrained manner, since it is not garbage collected.

A secondary benefit of stack allocation is the elimination of occasional synchronization for allocation of objects from the thread-common heap. In order to avoid synchronization on each heap allocation, the run-time system in HPCJ uses the following scheme. Each thread usually allocates objects from its thread-local heap space. For allocating a large object or when the local heap space is exhausted, the thread needs to allocate from thread-common heap space, which requires a relatively heavy-weight synchronization. Stack-allocated objects reduce the requirement for allocations from the thread-common heap space. Because the number of objects allocated from thread-common heap space, the benefits will vary depending on the particular allocation strategy employed by a particular JVM. Nevertheless, the information derived from escape analysis allows this benefit to be fully derived.

Elimination of synchronization operations requires run-time support at two places: allocation sites of objects, i.e., `new` sites; and use sites of objects as synchronization targets, i.e., `synchronized` methods or statements. In HPCJ, synchronized methods and statements are implemented using *monitorenter* and *monitorexit* atomic operations. The implementation of these operations in HPCJ has two parts: (1) atomic `compare_and_swap` operation for ensuring mutual exclusion, and (2) PowerPC `sync` primitive for flushing the local cache.

We mark objects at the allocation sites using a single bit in the object representation, indicating whether the object is thread-local. At the use sites of objects, we modified the routine implementing *monitorenter* on an object to bypass the expensive atomic operation (`compare_and_swap`) if its thread-local bit is set, and instead use a non-atomic operation. It is important to note that our scheme has benefits even for the thin-lock synchronization implementation [Bacon et al. 1998], which still needs an atomic operation (`compare_and_swap`); we completely eliminate the need for atomic lock operations for thread-local objects. Note that we still flush the local memory to ensure that global variables are made visible at synchronization points to observe Java semantics [Gosling et al. 1996]. Since the only change we make regarding synchronization is to eliminate the instructions that ensure mutual exclusion, the semantics of all other thread-related operations such as `wait` and `notify` remain unchanged as well.

Program	Number of objects allocated		Size of objects in bytes allocated		Total number of locks	
	user	user + library	user	user + library	user	user + library
trans	263K	727K	7656K	31333K	868K	885K
jgl	3808K	4157K	124409K	139027K	10391K	10434K
jacorb	103K	48036K	2815K	3423323K	546K	672K
jolt	94K	593K	3006K	17511K	1030K	1348K
jobe	204K	339K	7957K	13331K	77K	106K
javacup	67K	330K	1672K	8454K	191K	287K
hashjava	173K	248K	4671K	8270K	158K	165K
toba	154K	2201K	5878K	59356K	1060K	1246K
wingdis	840K	2561K	25902K	92238K	2105K	2299K
pbob	19787K	48206K	639980K	2749520K	35691K	171189K

Table II. Benchmarks characteristics

8. EXPERIMENTAL RESULTS

This section evaluates escape analysis on several Java benchmark programs. We experimented with four variants of the algorithm for the two applications: (1) Flow sensitive (FS) analysis, (2) Flow sensitive analysis with bounded field nodes (BFS), (3) Flow insensitive analysis (FI), and Flow insensitive analysis with bounded field nodes (BFI). The difference between FS and FI is that FI ignores the control-flow graph and never kills. Bounded field nodes essentially limit the number of field nodes that we wish to model for each object. We use a simple *mod* operation to keep the number of field nodes bounded. For instance, the kth reference field of an object can be mapped to the (k mod m)th field node. In our implementation, we used $m = 3$. Bounding the number of fields reduces the space and time requirement for our analysis, but can make the result less precise.

Our testbed consisted of a 333 MHz uniprocessor PowerPC running AIX 4.1.5, with 1 MB L2 Cache and 512 MB memory. We selected a set of 10 medium-sized to large-sized benchmarks described in Table I for our experiments. Columns 3 and 4 give the number of classes and the size of the classes in bytes for the set of programs. Table II gives the relevant characteristics for the benchmark programs. Columns 2 and 3 present the total number of objects dynamically allocated in the user code and overall (including both the user code and the library code). Columns 4 and 5 show the cumulative space in bytes occupied by the objects during program execution. Finally, columns 6 and 7 show the total number of lock operations dynamically encountered during execution.

In the rest of this section, we present our results for the above variants of our analysis. All of the remaining measurements that we present refer to objects created in the user code alone. Modifying any operations related to object creation in the library code would require recompilation of the library code (not done in our current implementation). Thus, our implementation analyzes library code while performing interprocedural analysis, but does not transform library code. Section 8.1 discusses results for stack allocation of objects. Section 8.2 discusses results for synchronization elimination. Section 8.3 discusses the actual execution time improvements due

to these two optimizations.

8.1 Stack Allocation

Figure 10 shows the percentage of user objects that we allocate on the stack, and Figure 11 gives the percentage in terms of space (bytes) that is stack-allocatable.

A substantial number of objects are stack-allocatable for `jacorb`, `jolt`, `wingdis`, and `toba` (if one does not bound the number of fields nodes). We did not see much difference between FS and FI (i.e. flow-sensitive and flow-insensitive without bounding the number of fields distinguished). And in most cases, bounding the number of field did not make much difference in the percentage values (for example, see `trans`, `jgl`, `jolt`, `jobe`, `javacup`, `hashjava`, and `wingdis`). Interestingly, `toba` and `jolt` (both of which are Java to C translators) have similar characteristics in terms of stack allocatability of objects. Both of these benchmarks have a substantial number of objects that are stack-allocatable. But in the case of `toba`, limiting the number of fields drastically reduces the number of objects that are stack-allocatable.

8.2 Lock Elimination

For lock elimination, we collected two sets of data (again for different variants of the analysis). We measured both the number of dynamic objects that are identified by our analysis as thread-local and how many lock operations are executed over these objects. Figure 12 shows the percentage of user objects that are determined to be local to a thread, and Figure 13 shows the percentage of lock operations that are removed for these thread-local objects during execution. It can be seen that our most precise analysis version finds a lot of opportunities to eliminate synchronization, removing more than 50% of the synchronization operations in half of the programs. One can deduce certain interesting characteristics by comparing the two graphs. For `pbob`, one can see that the percentage of thread-local objects ($\approx 50\%$) is higher than the percentage of locks removed ($\approx 15\%$). Our observation is that relatively few thread-local objects are actually involved in synchronization.

For `wingdis`, we have found a large percentage of objects that are thread-local ($\approx 75\%$), and were able to remove $\approx 91\%$ of synchronization operations. Notice that `jobe` has very few objects identified as thread-local. However, there are a relatively large number of synchronization operations performed on them, leading to a significant opportunity for eliminating those operations. The versions of our analysis using unbounded number of field nodes are able to remove a much higher percentage of synchronization operations than the bounded versions (even though the percentage of objects identified as thread-local, ranging from 0.3% to 0.8%, is too small to be noticeable). We conjecture that this difference comes from the fact that in the bounded cases, some *GlobalEscape* fields and *NoEscape* fields can be mapped onto the same node, resulting in loss of precision. Another interesting characteristic we observed is that for most cases, all four variants of the analysis performed equally well (except for `jacorb`, `hashjava`, `toba`, and `pbob`). For `toba`, bounding the number of fields, again, significantly reduced the percentage values of both the number of thread-local objects and the number of synchronization operations that could be eliminated.

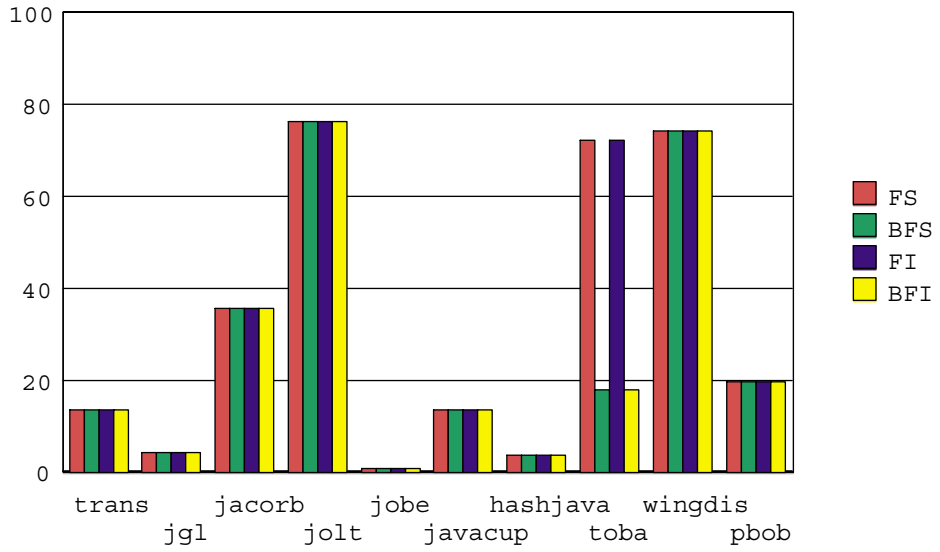


Fig. 10. Percentage of user code objects allocated on the stack.

8.3 Execution Time Improvements

Table III summarizes our results for execution time improvements. The second column shows the execution time (in seconds) prior to applying optimizations due to escape analysis. The third column shows the percentage reduction in execution

Program	Execution time (sec)	percentage reduction	potential of sync elimination
trans	5.2	7 %	2 %
jgl	18.8	23 %	5 %
jacorb	2.5	6 %	5 %
jolt	6.8	4 %	4 %
jobe	9.4	2 %	1 %
javacup	1.4	6 %	0 %
hashjava	6.4	5 %	2 %
toba	4.0	16 %	4 %
wingdis	18.0	15 %	2 %
pbob	N/A	6 %	N/A

Table III. Improvements in execution time

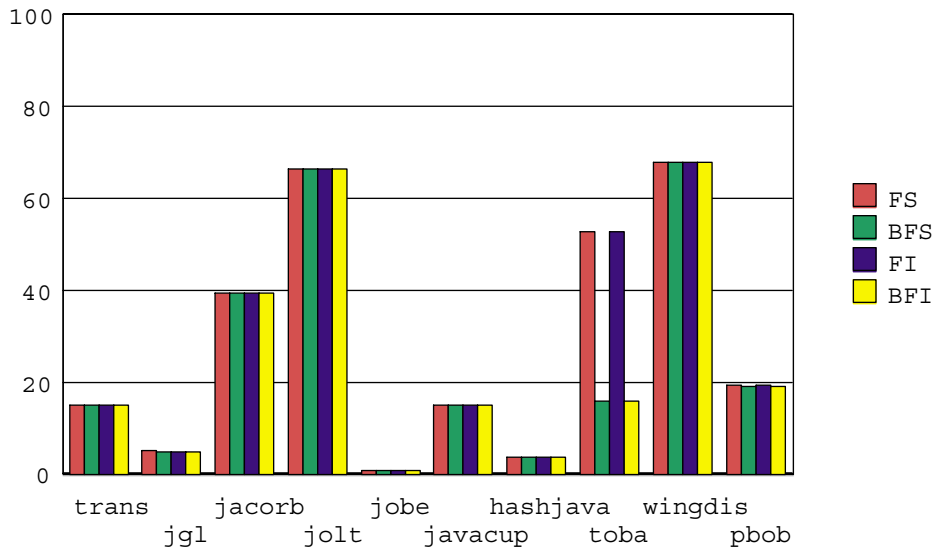


Fig. 11. Percentage of user code object space allocated on the stack.

time due to stack allocation of objects and synchronization elimination with our flow-sensitive analysis version. The time for `pbob` is not shown, because it runs for a predetermined length of time; its improvement is given as an increase in the number of transactions in that time period. `pbob` was run on a 4-way PowerPC SMP machine.

Table III shows an appreciable performance improvement (greater than 15% reduction in execution time) in three programs (`wingdis`, `jgl` and `toba`), and relatively modest improvements in other programs. `jgl` had a significant percentage of thread local objects, and a corresponding high percentage of locks removed, which contributed to its good performance. `wingdis` and `toba` shared these characteristics, and a substantial percentage of their objects were also stack allocatable. The table also shows the improvement that results from removing all `sync` instructions from the code. This gives an upper bound on the performance improvements that can be expected from implementing the *sync* removal analysis of Section 6. We note that the benefit is potentially greater for programs with threads executing on multiple processors since the overhead incurred by the *sync* instruction is greater.

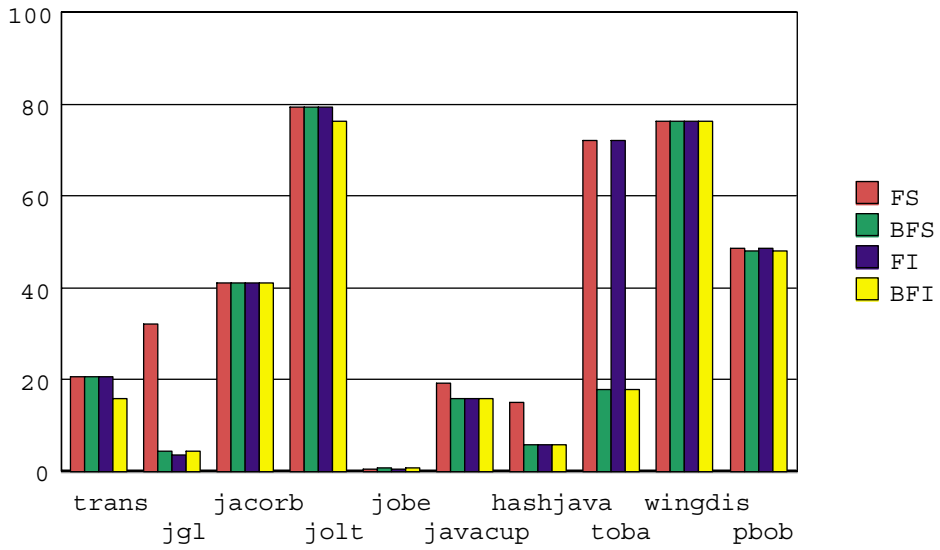


Fig. 12. Percentage of thread local objects in user code.

9. RELATED WORK

Lifetime analysis of dynamically allocated objects has been traditionally used for compile time storage management [Ruggieri and Murtagh 1988; Park and Goldberg 1992; Birkedal et al. 1996]. Park and Goldberg introduced the term *escape analysis* [Park and Goldberg 1992] for statically determining which parts of a list that are passed to a function do not escape the function call (and hence can be stack allocated). Others have improved and extended Park and Goldberg's work [Deutsch 1997; Blanchet 1998]. Birkedal et al. [Birkedal et al. 1996] propose a region allocation model, where regions are managed during compilation. A type system is used to translate a functional program to another functional program annotated with regions where values could be stored. Hannan [Hannan 1995] uses a type system to translate a strongly typed functional program to an annotated functional program, where the annotation is used for stack allocation rather than for region allocation. In our case we only stack allocates objects that do not escape a method and are created in the method. In some cases it is possible preallocate objects that escape a method akin to region allocation [Gay and Steensgaard 1999; Gay and Aiken 1998].

Prior work on synchronization optimization has addressed the problem of reduc-

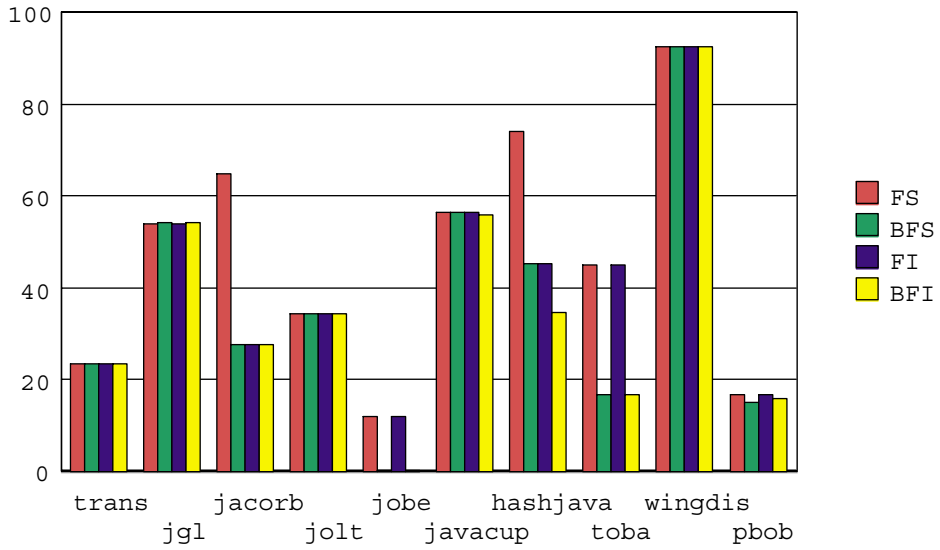


Fig. 13. Percentage of locks removed on objects in user code.

ing the amount of synchronization [Diniz and Rinard 1997; Li and Abu-Sufah 1987; Midkiff and Padua 1987]. These approaches assume that the mutual exclusion ordering implied by the original synchronization is needed, and so only attempt to reduce the number of such operations without violating the original ordering. In contrast, our approach finds unnecessary mutual exclusion lock operations and eliminates them.

There have been a number of parallel efforts on escape analysis for Java [Choi et al. 1999; Gay and Steensgaard 1999; Reid et al. 1999; Bodga and Hölzle 1999; Aldrich et al. 1999; Whaley and Rinard 1999; Blanchet 1999]. The current article is an extended version of Choi et al. [Choi et al. 1999], and includes a more detailed discussion of various optimizations and analysis.

Bogda and Hölzle use set constraints for computing thread-local objects [Bodga and Hölzle 1999]. Their system is a bytecode translator, and uses replication of execution paths as the means for eliminating unnecessary synchronization. After replication, they convert synchronized methods that access only thread-local objects into non-synchronized methods. This conversion, in general, breaks Java semantics—since at the beginning and the end of a synchronized method or a statement, the local memory has to be synchronized with the main memory (see

Section 7). Replication, however, offers an opportunity for specializing an allocation site that generates both thread-local and thread-global objects along different call chains. They also summarize the effect of native methods (although manually). Using the summary information, they improve the precision of their analysis. Our approach can be extended to include specialization and native method analysis.

Whaley and Rinard describe an escape analysis technique that is similar to ours. Two notable differences are that they construct a regular points-to graph and for certain cases they are able to perform strong updates on field variables [Whaley and Rinard 1999]. In a more recent paper, they use a demand-driven approach to speed up the analysis [Vivient and Rinard 2001].

Aldrich et al. describe a set of analyses for eliminating unnecessary synchronization on multiple re-entries of a monitor by the same thread, nested monitors, and thread-local objects [Aldrich et al. 1999]. They also remove synchronization operations, which can break Java semantics. They claim that their approach, however, should be safe for most well-written multithreaded programs in Java, which assume a "looser synchronization" model than what Java provides.

Blanchet uses type heights (which are integer values) to encode how an object of one type can have references to other objects or is a subtype of another object [Blanchet 1999]. The escaping part of an object is represented by the height of its type. He proposes a two-phase (a backward phase and a forward phase) flow-insensitive analysis for computing escape information. He uses escape analysis, like our work, for both stack allocation and synchronization elimination. For synchronization elimination, before acquiring a lock on an object o, his algorithm tests at runtime whether o is on the stack – if it is, the synchronization is skipped. Our algorithm uses a separate thread-local bit within each object, and can skip the synchronization even for objects that are not stack allocatable (but are thread local). In our approach when an object is reachable from a global variable we do not eliminate synchronization on it, since multiple threads may potential access it. Ruf's analysis essentially computes threads that access global escape objects [Ruf 2000]. If only one thread accesses such global escape objects, synchronization operations can be eliminated on them.

A key difference between the points-to graph used in pointer analysis and the connection graph used in escape analysis is that if two access paths in a points-to graph are disjoint, then the corresponding objects in the two paths do not interfere. On the other hand, if two paths are disjoint in the connection graph, nothing can be said about the interference of the objects in the access path. So our mapping of the callee connection graph with the caller connection graph is simpler. Fähndrich et al. also observe the above property and use directional information to speed-up queries on data flow information [Fähndrich et al. 2000].

To reduce the size of finite-state models of concurrent Java programs, Corbett uses a technique called virtual coarsening [Corbett 1998]. In virtual coarsening, invisible actions (e.g., updates to variables that are local or protected by a lock) are collapsed into adjacent visible actions. Corbett uses a simple intraprocedural pointer analysis (after method inlining) to identify the heap objects that are local to a thread, and also to identify the variables that are guarded by various locks.

Pugh describes some problems with the semantics of the Java memory model [Pugh 1999], and is spearheading an effort to revise the memory

model [JavaMemoryModel]. All memory flush operations associated with thread-local objects may be eliminated without further analysis (as described in Section 6) under one of the proposals being considered for the revised Java memory model [JavaMemoryModel].

Our connection graph abstraction is related to the alias graph and points-to graph proposed in the literature [Larus and Hilfinger 1988; Chase et al. 1990; Choi et al. 1993; Emami et al. 1994; Ghiya and Hendren 1998; Chatterjee et al. 1999]. Conventional representations for pointer analysis, including alias graph and points-to graph, are used for understanding memory disambiguation and cannot be easily summarized on a per procedure basis [Landi and Ryder 1992; Choi et al. 1993; Emami et al. 1994]. Researchers have, therefore, presented techniques to reduce the number of distinct calling contexts for which the procedure needs to be summarized for pointer analysis. These techniques include memoization to reuse the data flow solution for a given calling context [Wilson and Lam 1995; Ghiya and Hendren 1996], and inferring the relevant conditions on the unknown incoming context while summarizing a procedure [Chatterjee et al. 1999]. Compared to Wilson and Lam's approach to memoization, we do lose some precision since we do not reanalyze a method in cases where the resolution to a reference (i.e., function pointer in the case of Wilson and Lam) is subsequently made more precise.

The connection graph is a simpler abstraction than alias or points-to graph. We have shown that for identifying stack-allocatable objects, we can summarize the connection graph on a per procedure basis regardless of the incoming calling context, without losing precision. Hence, in this context, connection graph analysis is amenable to elimination-based data flow analysis [Marlowe 1989], i.e., one can construct a closure operation that essentially summarizes the effects of a method.

Ghiya and Hendren [Ghiya and Hendren 1996] introduce a related abstraction, the *connection matrix*, to determine whether an access path exists between the objects pointed to by two heap-directed pointers. They use this information for shape analysis of heap-allocated objects [Ghiya and Hendren 1996], and for memory disambiguation [Ghiya and Hendren 1998]. They do not use deferred edges in their representation, and have to compute the connection matrix for a procedure repeatedly, for different calling contexts, as their work targets a more general problem.

10. CONCLUSIONS

In this paper, we have presented a new interprocedural algorithm for escape analysis. Apart from using escape analysis for stack allocation of objects, we have demonstrated an important new application of escape analysis – eliminating unnecessary synchronization in Java programs. Our approach uses a data flow analysis framework and maps escape analysis to a simple reachability problem over a connection graph abstraction. With a preliminary implementation of this algorithm, which analyzes but does not transform class library code, our static Java compiler is able to detect a significant percentage of dynamically created objects in user code as stack-allocatable, as high as 70% in some cases. It is able to eliminate 11% to 92% of mutex lock operations (on objects created in the user code) in our benchmarks, eliminating more than 50% of those lock operations in half of the benchmarks. We observe overall performance improvements ranging from 2% to 23% on our

benchmarks, and find that most of these improvements come from savings on lock operations on the thread-local objects, as these programs do not seem to incur a significant garbage collection overhead due to relatively low memory usage.

We expect to improve these results in the future with a more aggressive implementation of our algorithm that treats native methods less conservatively, and by applying our optimizations to the Java standard class library routines as well. We also plan to implement our algorithm to eliminate unnecessary `sync` operations for flushing of local memory.

Interprocedural analysis in the presence of dynamic loading of classes, as allowed in Java, is in general a hard problem. We are currently working on extending our escape analysis to Jalapeño, a dynamic Java compilation system at IBM Research [Burke et al. 1999].

Acknowledgement

We would like to thank David Bacon, Michael Burke, Mike Hind, Ganesan Ramalingam, Vivek Sarkar, Ven Seshadri, Marc Snir, and Harini Srinivasan for useful technical discussions. We also thank OOPSLA'99 and PLDI'99 referees for their insightful comments on early drafts of the paper.

REFERENCES

Aldrich, J., Chambers, C., Sirer, E. G., and Eggers, S. 1999. Static analysis for eliminating unnessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium*, Venezia, Italy.

Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., and Zwaenepoel, W. 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer 29,* 2 (Feb.), 18–28.

Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. 1998. Thin locks: Featherweight synchronization for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada.

Birkedal, L., Tofte, M., and Vejlstrup, M. 1996. From region inference to von Neumann machines via region representation inference. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*.

Blanchet, B. 1998. Escape analysis: Correctness, proof, implementation and experimental results. In *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, pp. 25–37.

Blanchet, B. 1999. Escape analysis for object oriented languages: Application to Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado.

Bodga, J. and Hölzle, U. 1999. Removing unnecessay synchronization in Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado.

Burke, M., Carini, P., Choi, J.-D., and Hind, M. 1995. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.), *Lecture Notes in Computer Science, 892*, pp. 234–250. Springer-Verlag. Proceedings from the *7th Workshop on Languages and Compilers for Parallel Computing.* Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September 1994.

Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H., and Whaley, J. 1999. The Jalapeño dynamic optimizing compiler for Java. In *Proc. ACM SIGPLAN 1999 Java Grande Conference*.

Chambers, C., Pechtchanski, I., Sarkar, V., Serrano, M. J., and Srinivasan, H. 1999. Dependence analysis for Java. In *12th International Workshop on Languages and Compilers for Parallel Computing*.

Chase, D. R., Wegman, M., and Zadeck, F. K. 1990. Analysis of pointers and structures. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 296–310. *SIGPLAN Notices 25(6)*.

Chatterjee, R., Ryder, B. G., and Landi, W. A. 1999. Relevant context inference. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*.

Choi, J.-D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pp. 232–245.

Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. 1999. Escape analysis for Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado.

Corbett, J. C. 1998. Constructing compact models of concurrent Java programs. In *Proceedings of the 1998 International Symposium of Software Testing and Analysis*. ACM Press.

Deutsch, A. 1997. On the complexity of escape analysis. In *Proc. 24th Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, pp. 358–371.

Diniz, P. and Rinard, M. 1997. Synchronization Transformations for Parallel Computing. In *Proceedings of the 9'th Workshop on Languages and Compilers for Parallel Computers*.

Emami, M., Ghiya, R., and Hendren, L. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242–256.

Fähndrich, M., Rehof, J., and Das, M. 2000. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN'00 Conference of Programming Language Design and Implementation*, pp. 253–263.

Gay, D. and Aiken, A. 1998. Memory management with explicit regions. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada.

Gay, D. and Steensgaard, B. 1999. Stack allocating objects in Java. Research Report, Microsoft Research.

Ghiya, R. and Hendren, L. J. 1996. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming 24,* 6, 547–578.

Ghiya, R. and Hendren, L. J. 1998. Putting pointer analysis to work. In *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, pp. 121–133.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java$^{(TM)}$ Language Specification*. Addison-Wesley.

Goyal, D. 2000. A language-theoretic approach to algorithms. Ph.D. thesis, New York University. URL: http://cs.nyu.edu/phd_students/deepak/thesis.ps.

Gupta, M., Choi, J.-D., and Hind, M. 2000. Optimizing Java programs in the presence of exceptions. In *Proc. European Conference on Object-Oriented Programming*, Cannes, France. Also available as IBM T. J. Watson Research Center Tech. Report RC 21644.

Hannan, J. 1995. A type-based analysis for stack allocation in functional languages. In *Proc. 2nd International Static Analysis Symposium*.

Hind, M., Burke, M., Carini, P., and Choi, J.-D. 1999. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems 21,* 4 (July), 848–894. available as IBM Research Report RC8752.

International Business Machines Corporation 1997. IBM High Performance Compiler for Java. Available for download at `http://www.alphaWorks.ibm.com/formula`.

JavaMemoryModel. Java memory model mailing list. Archive at `http://www.cs.umd.edu/pugh/java/memoryModel/archive/`.

Landi, W. and Ryder, B. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248. *SIGPLAN Notices* 27(6).

Larus, J. R. and Hilfinger, P. N. 1988. Detecting conflicts between structure accesses. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 21–34. *SIGPLAN Notices, 23(7).*

Li, Z. and Abu-Sufah, W. 1987. On reducing data synchronization in multiprocessed loops. *IEEE Transactions on Computers C-36,* 1 (January), 105–109.

Marlowe, T. J. 1989. Data Flow Analysis and Incremental Iteration. Ph.D. thesis, Rutgers University.

May, C., Silha, E., Simpson, R., and Warren, H. (Eds.) 1994. *The PowerPC Architecture.* Morgan Kaufmann Publishers, Inc.

Midkiff, S. P. and Padua, D. A. 1987. Compiler algorithms for synchronization. *IEEE Transactions on Computing C-36,* 12 (Dec), 1485–1495.

Park, Y. and Goldberg, B. 1992. Escape analysis on lists. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 117–127.

Pugh, W. 1999. Fixing the Java memory model. In *ACM 1999 Java Grande Conference*, pp. 89–98.

Reid, A., McCorquodale, J., Baker, J., Hsieh, W., and Zachary, J. 1999. The need for predictable garbage collection. In *WCSSS'99 Workshop on Compiler Support for System Software.*

Ruf, E. 2000. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN'00 Conference of Programming Language Design and Implementation*, pp. 208–218.

Ruggieri, C. and Murtagh, T. 1988. Lifetime analysis of dynamically allocated objects. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 285–293.

Sagiv, M., Reps, T., and Wilhelm, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems 20,* 1 (Jan.), 1–50.

Seshadri, V. 1997. IBM high performance compiler for Java. *AIXpert Magazine*. Electronic publication available at URL `http://www.developer.ibm.com/library/aixpert`.

Vivient, F. and Rinard, M. 2001. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 Conference of Programming Language Design and Implementation.*

Whaley, J. and Rinard, M. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado.

Wilson, R. P. and Lam, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 1–12. *SIGPLAN Notices, 30(6).*

A. CORRECTNESS

In this section, we establish the correctness of our escape analysis algorithm. We will overload the notation $p.f_1.f_2 \ldots f_n$, called a *reference expression*, to denote a set of *points-to paths* of length n starting from a variable p, where $f_1, \ldots, f_n$ denote field nodes in the path. Let P denote a reference expression, then $PointsTo(P, G)$ denotes the set of abstract objects in G pointed to by the last field node of the paths represented by the reference expression. We use $PointsTo(P)$ as a shorthand for $PointsTo(P, G)$ when G is clear from the context. For object O, either an abstract or a concrete object, $AllocSite(O)$ denotes the set of allocation sites for O. For a concrete object, $AllocSite(O)$ will be a singleton, consisting of a single allocation site.

When a field of an object is read- or write-accessed in method M, the object might be the one created by a caller of M. We call such objects *upwards-exposed objects*. A more formal definition is as follows:

Definition A.1. Object O is *upwards-exposedly accessed* at program point pp in M if (1) a field of O is read-accessed or write-accessed at pp, and (2) there exists a path in the control flow graph from the entry of M to pp along which O is not created. If O is upwards-exposedly accessed at a program point in M, O is an *upwards-exposed object*.

The main theorem that establishes the correctness of escape analysis is Theorem A.18. Intuitively, Theorem A.18 states that (1) if an allocation site A is marked *NoEscape*, then all concrete objects allocated at this site are local to the method, and (2) if it is marked *ArgEscape*, then all concrete objects allocated at this site are local to the thread. To prove the main theorem requires several supporting lemmas.

In the following, we use G^c to denote the concrete storage graph created during the execution of a program, G^a to denote the *points-to alias graph* of the abstract objects and edges similar to those proposed in the literature [Choi et al. 1993; Hind et al. 1999], and implemented, for computing points-to aliases of the same program, and CG to denote the connection graph we build for the same program. We refer to an alias graph or a connection graph as an *abstract graph*.

According to the uniqueness property defined in Section 2, between G^c and G^a of a same program, for every concrete object $O_c \in G^c$, there exists a unique abstract object $O_a \in G^a$ that O_c can be mapped to. The points-to analysis of Choi et al. [Choi et al. 1993; Hind et al. 1999] relies on the uniqueness property for constructing points-to graph. In contrast to the uniqueness property, a concrete object can be mapped to multiple abstract objects in the connection graph, which we call the non-uniqueness property. This non-uniqueness property of the connection graph allows for efficient bottom-up computation of connectivity for escape analysis.

The non-uniqueness property, however, makes it impossible to straightforwardly apply proofs based on the uniqueness property to the correctness proof of our escape analysis based on the connection graph. This non-uniqueness make CG different from the G^c built for the same program. We, therefore, take a three-step approach to prove the correctness of our approach. The first step is to establish that computing the reachability of abstract objects over G^a is a safe approximation of computing the reachability of concrete objects over G^c. We express this as $G^c \sqsubseteq G^a$. We establish this by showing that for every access path to object $O_c \in G^c$ at a program point there is a points-to path to object $O_a \in G^a$ at that point such that $AllocSite(O_c) \subseteq AllocSite(O_a)$. The second step is to establish that our way of computing the reachability of abstract objects over CG is equivalent to computing the reachability of abstract objects over G^a. We express this as $G^a \sqsubseteq CG$. (Actually, $G^a \sqsubseteq CG$ and $CG \sqsubseteq G^a$ both hold.) From these two steps, we can trivially establish that $G^c \sqsubseteq CG$: our way of computing the reachability of abstract objects over CG is a safe approximation of computing the reachability of concrete objects over G^c.

Finally we show, based on $G^c \sqsubseteq CG$, that our way of identifying local and escaping objects using the connection graph is correct. The second step of establishing $G^a \sqsubseteq CG$ is necessary due to the non-uniqueness property of CG, and, therefore, is the focus of our correctness proof.

A.1 $G^c \sqsubseteq G^a$

We first state the following two properties of G^a (whose proofs are beyond the scope of this paper [Choi et al. 1993; Hind et al. 1999].)

PROPERTY 2. *Every concrete node $c_i \in G^c$ is mapped to a single abstract node $a_j \in G^a$ such that $AllocSite(c_i) \subseteq AllocSite(a_j)$, expressed as $a_j = MapCtoA(c_i)$.*

PROPERTY 3. *Let $c_i, c_j \in G^c$, $a_m = MapCtoA(c_i), a_n = MapCtoA(c_j)$, and f be a field of c_i. Then, $c_i.f \rightarrow c_j$ only if $a_m.f \rightarrow a_n$.*[7]

Property 2 and Property 3 hold for points-to graphs that are constructed using the algorithm given in [Choi et al. 1993; Hind et al. 1999], which employ k-limited creation of abstract objects.[8]

Given the above properties we can establish the following theorem.

LEMMA A.2. *Let $O_c \in G^c$ and $O_a \in G^a$. There exist a set of paths $P_c = p.f_1.f_2 \ldots f_n \rightarrow O_c \in G^c$ only if there exist a set of paths $P_a = p.f_1.f_2 \ldots f_n \rightarrow O_a \in G^a$ such that $AllocSite(O_c) \subseteq AllocSite(O_a)$.*

PROOF. By induction on path length, based on Property 3. □

THEOREM A.3. $G^c \sqsubseteq G^a$

PROOF. According to Lemma A.2. □

A.2 $G^a \sqsubseteq CG$

We now establish that our way of computing the reachability of objects over CG is equivalent to computing that over G^a. As noted previously, CG contains phantom nodes (i.e. objects accessed in a method in which they are not created) that introduce extra paths compared to G^a. We therefore have to show that these extra paths will not affect the escape state of objects.

We use a_i to denote a *phantom reference variable* (corresponding to a phantom reference node) holding the initial value of formal parameter f_i. We call a_i the actual parameter at the callee of formal parameter f_i. We use $\hat{a}_i$ to denote the (temporary) variable of the caller holding the value of the actual parameter corresponding to a_i (and, thereby, to f_i) of the callee. We call $\hat{a}_i$ the actual parameter at the caller of f_i. The pair of a_i and $\hat{a}_i$ is used to identify nodes and edges, in the caller and the callee, that correspond to each other. If the method returns a reference (i.e. pointer in Java) value, the returned expression is treated as an assignment to an actual parameter corresponding to the return, and the method call is treated as read-accessing the actual parameter at the caller.

First a few more definitions are needed. *Local variables* are named memory locations in Java that are defined in a method and that can be accessed only within

[7]For efficiency, sometimes multiple fields of a concrete node are mapped to a single edge of the corresponding abstract node. In this case, the property can be stated as: $c_i.f \rightarrow c_j$ only if $a_m.g \rightarrow a_n$, where g is the field that f is mapped to. However, ignoring this does not affect the correctness arguments presented herein.

[8]Others [Larus and Hilfinger 1988; Chase et al. 1990; Ghiya and Hendren 1998; Chatterjee et al. 1999], have proposed points-to alias graphs which are slight variants of the one proposed in [Hind et al. 1999; Choi et al. 1993].

a method. Local variables include the formal parameters $f_i, 1 \leq i \leq n$. If the method is a virtual method, formal parameter f_0 denotes the `this` pointer. *Non-local variables* are named memory locations in Java, such as static field of a class, that can be accessed by more than one method. Actual parameters are regarded as written at a call site, and are read within a called method initially through the formal parameters. Actual parameters are, therefore, regarded as non-local variables since they can be accessed by a caller and a callee. An object is local to a method if there exists no path from a non-local variable of the method to the object. Otherwise, it is a non-local object. We call the maximal subgraph of G not reachable from any non-local the *local subgraph* of G.[9]

We create at most one abstract object per statement, and name each abstract object after its allocation site. A Java statement containing more than one creation sites of abstract objects is broken into smaller statements by the Java bytecode generator. Therefore, we can identify whether an object in G^a and an object in CG correspond to each other by their allocation sites.

Definition A.4. For $O_a \in G^a$ and $O_{cg} \in CG$, we say $O_a \cong O_{cg}$ if and only if $AllocSite(O_a) = AllocSite(O_{cg})$.

Definition A.5. G_1 and G_2 are congruent, denoted as $G_1 \cong G_2$, if for each reference expression P, $PointsTo(P, G_1) = PointsTo(P, G_2)$. Note that $((G_1 \cong G_2) \wedge (G_2 \cong G_3)) \Longrightarrow (G_1 \cong G_3)$

Let G_{cs} and G_{ee} be the abstract graph holding at callsite cs and the abstract graph of the callee invoked at cs, respectively.[10]

Definition A.6. We use $G_{cs} \oplus G_{ee}$ to denote the application of `UpdateCaller()` to abstract graphs G_{cs} and G_{ee}.

For simplicity of discussion, we assume routine `UpdateCallerNodes()` and `UpdateCallerEdges()` in Section 4 are applied for all non-locals. We also assume that these routines also propagate the local subgraph of the callee to the callsite graph such that for each edge from a local object O_l to a non-local object O_n in the callee graph there exist a set of edges from O_l to each of the objects to which O_n is mapped to in the callsite graph.[11]

Let G^a_{cs} be the points-to graph holding at callsite cs invoking method M, G^a_i be the points-to graph holding at program point pp_i in M with respect to callsite cs, and CG_i be the connection graph holding at pp_i. A major purpose of this section (Section A.2) is to prove Theorem A.12:

$$G^a_i \cong G^a_{cs} \oplus CG_i,$$

[9]It can be a set of disjoint graphs.

[10]Without loss of generality, we assume each callsite uniquely identifies a single method. When multiple methods are identified as static targets of a method invocation at a callsite due to indirect or dynamic dispatch, a dataflow *meet* operation is performed over the results, one from each target.

[11]We assume that all the upwards-exposed accessed objects of a callee have correspondingly mapped objects in the points-to graph holding at a callsite of the callee. This is a reasonable assumption for a static analysis. One way to handle when the assumption breaks within a method is to ignore the CFG edge where it breaks until the analysis converges or an object becomes reachable along a CFG edge.

based on which we prove $G^a \sqsubseteq CG$. Note that CG_i, as a connection graph, has the non-uniqueness property and might have phantom nodes for upwards-exposed objects accessed in M. However, $G^a_{cs} \oplus CG_i$ results in a points-to graph, which has the uniqueness property. It does not have any phantom nodes, either.

Definition A.7. For objects O_s and O_t, we use $O_s.f_i \rightarrow O_t$ and $O_s \xrightarrow{f_i} O_t$ interchangeably to denote that there exists an edge from f_i field of O_s to O_t.

Definition A.8. Let $G_r = G_{cs} \oplus G_{ee}$. Then, for object $O_e \in G_{ee}$ and $O_r \in G_r$, $MapsToObj(O_e, G_{ee}, G_r) = \{O_r \mid O_e \longmapsto O_r\}$, where $\longmapsto$ is $MapsTo$ relation defined in Section 4. We use $MapsToObj(O_e)$ and $MapsToObj(O_e, G_r)$ as shorthands for $MapsToObj(O_e, G_{ee}, G_r)$ where the meaning is clear from the context.

LEMMA A.9. *Let $G_r = G_{cs} \oplus G_{ee}$. Then,*

$$O_1 \xrightarrow{f} O_2 \in G_{ee} \Longrightarrow$$
$$\{\hat{O}_1 \xrightarrow{f} \hat{O}_2\} \in G_r, \forall \hat{O}_1 \in MapsToObj(O_1, G_r), \forall \hat{O}_2 \in MapsToObj(O_2, G_r).$$

PROOF. By definition of $MapsToObj()$ and construction of `UpdateCaller()` in Section 4. □

For a set of objects S_o, we define $MapsToObj(S_o)$ as follows:

$$MapsToObj(S_o) = \bigcup_O MapsToObj(O), O \in S_o.$$

LEMMA A.10. *Let $G_o = G_{cs} \oplus G_{ee}$, and P be a reference expression. Then, $PointsTo(P, G_o) = MapsToObj(PointsTo(P, G_{ee}))$.*

PROOF. By induction on path lengths based on Lemma A.9. □

The following lemma establishes the associativity of the $\oplus$ operator. We use the associativity of $\oplus$ in establishing the relation between the connection graph holding right after a callsite invoking a method and the connection graph holding at the end of the called method. Note that both the left hand side and the right hand side of the $\oplus$ operation in Lemma A.11 result in points-to graphs.

LEMMA A.11. *Let G_a be the points-to graph holding at a callsite invoking method M, G_b be the connection graph holding at a callsite invoking N from within M, and G_c be the connection graph at program point pp_i within N. Then,*

$$(G_a \oplus G_b) \oplus G_c \cong G_a \oplus (G_b \oplus G_c).$$

PROOF. Let $G_0 = (G_a \oplus G_b) \oplus G_c$, and $G_1 = G_a \oplus (G_b \oplus G_c)$. We prove by showing that for all path expression P, $PointsTo(P, G_0) = PointsTo(P, G_1)$ by induction on path length. Note that we only need to consider paths that are not in G_a since $\oplus$ operation deletes neither nodes nor edges..

—**Base Case:**
(1) Consider objects $O^p_c, O^q_c \in G_c$ such that

$$g_p : \{p \rightarrow O^p_c\}, g_q : \{q \rightarrow O^q_c\}, g_{pq} : \{O^p_c \xrightarrow{f_1} O^q_c\} \in G_c. \quad (6)$$

We will first consider $G_a \oplus (G_b \oplus G_c)$.

Let

$$OP_b = MapsToObj(O_c^p, G_b \oplus G_c) = PointsTo(p, G_b) \cup \{O_c^p\}, \quad (7)$$
$$OQ_b = MapsToObj(O_c^q, G_b \oplus G_c) = PointsTo(q, G_b) \cup \{O_c^q\}.^{12} \quad (8)$$

The union of callee objects, such as O_c^p and O_c^q, with the caller objects happens only when the callee object is a non-phantom node or does not have any target objects of $MapsToObj()$ in the caller graph. However, all the target objects, including the callee object, will have the same outgoing edges, and regarding the callee object as being always unioned is conceptually easier to understand.
By Lemma A.9,

$$\{O_c^p \xrightarrow{f_1} O_c^q\} \in G_c \Longrightarrow \{O_b^p \xrightarrow{f} O_b^q\} \in G_b \oplus G_c, \forall O_b^p \in OP_b, \forall O_b^q \in OQ_b. \quad (9)$$

Let

$$\begin{aligned} OP_a &= MapsToObj(O_b^p, G_a \oplus (G_b \oplus G_c)), \forall O_b^p \in OP_b \\ &= PointsTo(p, G_a) \cup OP_b \\ &= PointsTo(p, G_a) \cup PointsTo(p, G_b) \cup \{O_c^p\} \quad (10) \\ OQ_a &= MapsToObj(O_b^q, G_a \oplus (G_b \oplus G_c)), \forall O_b^q \in OQ_b \\ &= PointsTo(q, G_a) \cup OQ_b \\ &= PointsTo(q, G_a) \cup PointsTo(q, G_b) \cup \{O_c^q\}. \quad (11) \end{aligned}$$

By Lemma A.9,

$$\{O_b^p \xrightarrow{f_1} O_b^q\} \in G_b, O_b^p \in OP_b, O_b^q \in OQ_b \Longrightarrow \{O_a^p \xrightarrow{f} O_a^q\} \in G_a \oplus (G_b \oplus G_c), \forall O_a^p \in OP_a, \forall O_a^q \in OQ_a, \quad (12)$$

which, from Equation (10) and Equation (11), becomes

$$\begin{aligned} \{O_a^p \xrightarrow{f} O_a^q\} &\in G_1, \\ &\forall\ O_a^p \in PointsTo(p, G_a) \cup PointsTo(p, G_b) \cup \{O_c^p\}, \\ &\forall\ O_a^q \in PointsTo(q, G_a) \cup PointsTo(q, G_b) \cup \{O_c^q\}. \quad (13) \end{aligned}$$

We now consider $(G_a \oplus G_b) \oplus G_c$.
Let

$$\hat{O}_b^p \in PointsTo(p, G_b), \hat{O}_b^q \in PointsTo(q, G_b).$$

By Lemma A.9,

$$MapsToObj(\hat{O}_b^p, G_a \oplus G_b) = PointsTo(p, G_a) \cup \{\hat{O}_b^p\},$$
$$MapsToObj(\hat{O}_b^q, G_a \oplus G_b) = PointsTo(q, G_a) \cup \{\hat{O}_q^p\}.$$

Therefore,

$$\begin{aligned}
PointsTo(p, G_a \oplus G_b) &= \bigcup_{\hat{O}_b^p} MapsToObj(\hat{O}_b^p, G_a \oplus G_b) \\
&= PointsTo(p, G_a) \cup PointsTo(p, G_b), \\
PointsTo(q, G_a \oplus G_b) &= \bigcup_{\hat{O}_b^q} MapsToObj(\hat{O}_b^q, G_a \oplus G_b) \\
&= PointsTo(q, G_a) \cup PointsTo(q, G_b).
\end{aligned}$$

By Lemma A.9,

$$\begin{aligned}
&\{O_c^p \xrightarrow{f_1} O_c^q\} \in G_c \Longrightarrow \\
&\qquad \{\hat{O}_c^p \xrightarrow{f_1} \hat{O}_c^q\} \in (G_a \oplus G_b) \oplus G_c, \\
&\qquad\qquad \forall \hat{O}_c^p \in PointsTo(p, G_a \oplus G_b) \cup \{O_c^p\}, \\
&\qquad\qquad \forall \hat{O}_c^q \in PointsTo(q, G_a \oplus G_b) \cup \{O_c^q\} \\
&\qquad = \{\hat{O}_c^p \xrightarrow{f_1} \hat{O}_c^q\} \in G_0, \\
&\qquad\qquad \forall \hat{O}_c^p \in PointsTo(p, G_a) \cup PointsTo(p, G_b) \cup \{O_c^p\}, \\
&\qquad\qquad \forall \hat{O}_c^q \in PointsTo(q, G_a) \cup PointsTo(q, G_b) \cup \{O_c^q\}. \quad (14)
\end{aligned}$$

Equation 13 and Equation 14 are identical, and, therefore, the following holds for the base case:

$$(G_a \oplus G_b) \oplus G_c \cong G_a \oplus (G_b \oplus G_c).$$

(2) Consider objects $O_b^p, O_b^q \in G_b$ such that

$$g_p : \{p \rightarrow O_b^p\}, g_q : \{q \rightarrow O_b^q\}, g_{pq} : \{O_b^p \xrightarrow{f_1} O_b^q\} \in G_b.$$

$G_b \oplus G_c$ does not affect any of the following:

$$g_p : \{p \rightarrow O_b^p\}, g_q : \{q \rightarrow O_b^q\}, g_{pq} : \{O_b^p \xrightarrow{f_1} O_b^q\} \in G_b,$$

and, therefore,

$$\begin{aligned}
PointsTo(p, G_0) &= PointsTo(p, G_a \oplus (G_b \oplus G_c)) = PointsTo(p, G_a \oplus G_b) \\
&= PointsTo(p, G_a) \cup \{O_b^p\}, \\
PointsTo(p, G_1) &= PointsTo(p, (G_a \oplus G_b) \oplus G_c) = PointsTo(p, G_a \oplus G_b) \\
&= PointsTo(p, G_0), \\
PointsTo(q, G_0) &= PointsTo(q, G_a \oplus (G_b \oplus G_c)) = PointsTo(q, G_a \oplus G_b) \\
&= PointsTo(q, G_a) \cup \{O_b^q\}, \\
PointsTo(q, G_1) &= PointsTo(q, (G_a \oplus G_b) \oplus G_c) = PointsTo(q, G_a \oplus G_b) \\
&= PointsTo(q, G_0).
\end{aligned}$$

By Lemma A.9,

$$\{O_b^p \xrightarrow{f_1} O_b^q\} \in G_b \Longrightarrow$$
$$= \{\hat{O}_b^p \xrightarrow{f_1} \hat{O}_b^q\} \in G_0, G_1$$
$$\forall \hat{O}_b^p \in PointsTo(p, G_a) \cup \{O_b^p\},$$
$$\forall \hat{O}_b^q \in PointsTo(q, G_a) \cup \{O_b^q\}. \quad (15)$$

Therefore, the following holds for the base case:

$$(G_a \oplus G_b) \oplus G_c \cong G_a \oplus (G_b \oplus G_c).$$

—**Induction:**
The proof for the base case above can be extended to the general case by replacing objects reachable from a single variable reference expression of p and q to objects with objects reachable from reference expressions $p.f_1 \dots f_m$ and $q.g_1 \dots f_m$. For example, Equation (6), Equation (7), and Equation (8) become as follows:

$$g_p : \{\ p.f_1\ \dots f_m \rightarrow O_c^p\}, g_q : \{q.g_1 \dots g_n \rightarrow O_c^q\}, g_{pq} : \{O_c^p \xrightarrow{f_1} O_c^q\} \in G_c \quad (16)$$
$$OP_b = MapsToObj(O_c^p, G_b \oplus G_c)$$
$$= PointsTo(p.f_1 \dots f_m, G_b) \cup \{O_c^p\} \quad (17)$$
$$OQ_b = MapsToObj(O_c^q, G_b \oplus G_c)$$
$$= PointsTo(q.g_1 \dots g_n, G_b) \cup \{O_c^q\}. \quad (18)$$

With that modification, by following the same logic, we can show that the following holds in the general case:

$$(G_a \oplus G_b) \oplus G_c \cong G_a \oplus (G_b \oplus G_c).$$

□

THEOREM A.12. *Let G_{cs}^a be the points-to graph holding at callsite cs invoking a method M, G_i^a be the points-to graph holding at program point pp_i in M, with respect to cs, by propagating G_{cs}^a to the entry of M, and CG_i be the connection graph holding at pp_i. Then, $G_i^a \cong G_{cs}^a \oplus CG_i$.*

PROOF. We prove it by induction on program points $pp_i, 0 \leq i \leq n$, where pp_n is the exit of M.

—**Base Case:** This is the case at the entry of a method. Since $G_0^a = G_{cs}^a$ and $G_{cs}^a \oplus CG_0 = G_{cs}^a$, the induction hypothesis holds.

—**Induction:** Assuming $G_i^a \cong G_{cs}^a \oplus CG_i$ (right after pp_i), we show that $G_{i+1}^a \cong G_{cs}^a \oplus CG_{i+1}$ (right after pp_{i+1}). Recall that by definition, $G_1 \cong G_2 \Longleftrightarrow \forall P \in G_1, PointsTo(P, G_1) = PointsTo(P, G_2)$. There are six types of statements between pp_i and pp_{i+1} for a leaf method as follows:

—`p = new T:` (See Figure 5(1).) In the connection-graph, object O_{cg} is created, and subgraph $\zeta_{cg} = \{p \rightarrow O_{cg}\}$ is added to CG_i to construct CG_{i+1}. This results in $\zeta_{cg} \in (G_{cs}^a \oplus CG_{i+1})$.
In the points-to graph, object O_a is created with the same name (based on the allocation site) as that of O_{cg}, and subgraph $\zeta_a = \{p \rightarrow O_a\}$ is added to G_i^a. This results in $\zeta_a \in G_{i+1}^a$, where O_a and O_{cg} have identical names.

With killing, we assume p is a local variable of M since we only kill local variables (not including the case of "`p.g = q`"). Since p is a local variable of M, there is no path from p in G^a_{cs}, which holds at a callsite of M.[13] Therefore, there exists edge $E : p \rightarrow O_m \in (G^a_{cs} \oplus CG_i)$ only if there exists edge $E : p \rightarrow O_m \in CG_i$, and deleting all the edges $\{p \rightarrow O_m \mid O_m \in PointsTo(p, CG_i))\}$ from CG_i to construct CG_{i+1} results in deleting all the edges $\{p \rightarrow O_m \mid O_m \in PointsTo(p, G^a_i \oplus CG_i)\}$. On points-to graph G^a_i, deleting edges from p is deleting all the edges $\{p \rightarrow O_n \mid O_n \in PointsTo(p, G^a_i)\}$, which is the same as deleting all the edges $\{p \rightarrow O_n \mid O_n \in PointsTo(p, G^a_{cs} \oplus CG_i)\}$, by the induction hypothesis of $G^a_i \cong C^a_{cs} \oplus CG_i$. Therefore, the same set of edges are deleted from G^a_i and $G^a_{cs} \oplus CG_i$ to construct G^a_{i+1} and $G^a_{cs} \oplus CG_{i+1}$, and the induction holds.

Since it does not delete any edges that would have not been deleted in any previous iterations at the same statement, the transfer function of this statement is monotonic.

—`p = q:` (See Figure 5(2).) By the induction hypothesis and by Lemma A.10,

$$\begin{aligned} PointsTo(q, G^a_i) &= PointsTo(q, G^a_{cs} \oplus CG_i) \\ &= MapsToObj(PointsTo(q, CG_i), G^a_i). \end{aligned} \tag{19}$$

To G^a_i, we add the following edges to construct G^a_{i+1}:

$$p \rightarrow O_a, \forall O_a \in PointsTo(q, G^a_i). \tag{20}$$

To CG_i, we add the following edges to construct CG_{i+1}:

$$p \rightarrow O_{cg}, \forall O_{cg} \in PointsTo(q, CG_i). \tag{21}$$

According to Lemma A.10, adding edges in Equation (21) results in adding the following edges to $G^a_{cs} \oplus CG_i$:

$$p \rightarrow \hat{O}_{cg}, \forall \hat{O}_{cg} \in MapsToObj(PointsTo(q, CG_i), G^a_{cs} \oplus CG_i), \tag{22}$$

which, from Equation (19), is the same as follows:

$$p \rightarrow \hat{O}_{cg}, \forall \hat{O}_{cg} \in PointsTo(q, G^a_i). \tag{23}$$

Equation (23) is the same as Equation 20, which reflects what we perform to construct CG^a_{i+1}.

With killing, the same argument for "`p = new T;`" applies, and the operation on connection-graph building is monotonic.

—`p = q.f:` (See Figure 5(4).) When q has a target object in CG_i, the proof for the case of "`p = q`" can be used for this case by replacing q with $q.f$. For example, Equation (19) for induction hypothesis becomes as follows:

$$\begin{aligned} PointsTo(q.f, G^a_i) &= PointsTo(q.f, G^a_{cs} \oplus CG_i) \\ &= MapsToObj(PointsTo(q.f, CG_i)). \end{aligned} \tag{24}$$

Therefore, we only need to consider when q does not have a target object, resulting on construction of phantom nodes:

$$PointsTo(q, CG_i) = PointsTo(q.f, CG_i) = \emptyset. \tag{25}$$

[13] This is true even when M is called multiple times in a same method due to scoping rule.

Note that in this case,

$$PointsTo(q.f, G^a_i) = PointsTo(q.f, G^a_{cs} \oplus CG_i) = PointsTo(q.f, G^a_{cs}). \quad (26)$$

Without loss of generality, we assume q is a non-local (or there exists a path of only deferred edges from q to a non-loal variable).[14]
In the connection graph, we create two phantom nodes, O_1 and O_2, and add them to CG_i to construct CG_{i+1} such that $O_1 = PointsTo(q, CG_{i+1})$ and $O_2 = PointsTo(q.f, CG_{i+1})$. Then, we insert edge $p \rightarrow PointsTo(q.f, CG_{i+1})$, which, after performing `UpdateCaller()`, is converted into the following edges:

$$\begin{aligned}
& p \rightarrow O_{cg}, \forall O_{cg} \in PointsTo(q.f, G^a_{cs} \oplus CG_{i+1}) && (27)\\
=\ & p \rightarrow O_{cg}, \forall O_{cg} \in MapsToObj(PointsTo(q.f, CG_{i+1})) && (28)\\
=\ & p \rightarrow O_{cg}, \forall O_{cg} \in MapsToObj(PointsTo(q.f, CG_i) \cup \{O_2\}) && (29)\\
=\ & p \rightarrow O_{cg}, \forall O_{cg} \in MapsToObj(\{O_2\}) && (30)\\
=\ & p \rightarrow O_{cg}, \forall O_{cg} \in PointsTo(q.f, G^a_{cs}) && (31)\\
=\ & p \rightarrow O_{cg}, \forall O_{cg} \in PointsTo(q.f, G^a_i). && (32)
\end{aligned}$$

Equations (30) and (31) are derived from Equation (25), and Equation (32) is derived from Equation (26). The edges in Equation (32) are exactly the edges to be added to construct points-to graph G^a_{i+1}.
With killing, the same argument for "`p = new T;`" applies, and the operation on connection-graph building is monotonic.

—`p.g = q;` (See Figure 5(3).) It is similar to the case of "`p = q.f`" except that a phantom node might be created as a target of p, instead of q.
No killing happens with this statement, and the operation on connection-graph building is monotonic.

—`call N();` Let CG_X be the connection graph holding at the exit of N, and G^a_X be the points-to graph holding at the exit of N. Note that $CG_{N+1} = CG_N \oplus CG_X$ since our connection graph construction algorithm applies `UpdateCaller()` to the connection graph holding at the callsite (i.e., CG_N) and to the connection graph holding at the end of the callee (i.e., CG_X) in order to compute the connection graph holding right after the callsite (i.e., CG_{N+1}).

$$\begin{aligned}
G^a_{N+1} &= G^a_X \\
&\cong G^a_N \oplus CG_X && \text{(according to Induction Hypothesis)}\\
&\cong (G^a_M \oplus CG_N) \oplus CG_X && \text{(according to Induction Hypothesis)}\\
&\cong G^a_M \oplus (CG_N \oplus CG_X) && \text{(according to Lemma A.11)}\\
&\cong G^a_M \oplus CG_{N+1} && \text{(by connection graph construction)}
\end{aligned}$$

The last step comes from that we construct CG_{N+1} by $CG_N \oplus CG_X$. When a callsite has multiple static targets, we perform union of G^a_X by each method. If a target does not have its connection graph built yet, we optimistically assume

[14]Otherwise, q is a local pointing to nothing. This is a symptom of either a definite null-pointer error or of not-yet converged analysis due to control-flow graph back edges. In the latter, we expect to visit this statement again later with q pointing to a target.

its connection graph to be empty. The operation at a callsite, therefore, is monotonic.

—`control flow graph join:` We perform union of the connection graphs propagated from each of the incoming edges of the join node. For an incoming edge that has not been visited during intraprocedural iteration, we optimistically assume the connection graph from the edge to be empty. The operation at a join node, therefore, is monotonic.

Since all the operations are monotonic in building the connection graph, after convergence, $G_i^a \cong G_{cs}^a \oplus CG_i$ holds at every program point.

□

The connection graphs in Figure 4 illustrate Theorem A.12. The points-to graph holding at statement `S4` is congruent with the result of $\oplus$ applied to the connection graph of Figure 4(F) as G_{cs}^a and the connection graph of Figure 4(D) as CG_i. (After a bypass operation is applied to the deferred edge from `y` to `x`, the connection graph of Figure 4(F) can be regarded as the points-to graph holding at the callsite if no aliasing holds at the entry of the caller, `L()`.) Figure 4(G) shows the result of $\oplus$ operation. (Figure 4(G) also corresponds to a points-to graph after a bypass is applied.)

Definition A.13. The local subgraph of a points-to graph G^a for a method M, denoted as $G_l^a(M)$ (or simply G_l^a, where M is clear from the context), is the maximal subgraph of G^a obtained by removing all the nodes that are reachable from a non-local variable with respect to M, and by removing the edges incident to/from those nodes.

The lack of uniqueness property of a connection graph is due to the paths in the concrete graph and the abstract graph holding at a callsite of a method. These paths are reachable from the non-locals of a method. Since the local subgraph of the connection graph of a method is not reachable from any non-locals of the method, it does have the uniqueness property. Corollary A.14 states that the local subgraphs of the points-to graph and the connection graph of a method are congruent with each other.

COROLLARY A.14. *Let G_i^a and CG_i be the points-to graph and the connection graph, both holding at program point pp_i in method M, G_l^a be the local subgraph of G_i^a, and CG_l be the local subgraph of CG_i. Then, $G_l^a \cong CG_l$.*

PROOF. Since CG_l is not reachable from any non-local variables of M, CG_l will be just propagated, by `UpdateCaller()`, in computing $G_{cs}^a \oplus CG_i$. Since $G_i^a \cong G_{cs}^a \oplus CG_i$, $PointsTo(P, G_l^a) = PointsTo(P, CG_l)$ for any path P starting with a local variable of M. Therefore, $G_l^a \cong CG_l$, □

THEOREM A.15. *Let G_i^a and CG_i be the points-to graph and the connection graph, both holding at program point pp_i in method M, G_l and CG_l be the local subgraphs of G_i^a and CG_i, respectively, and O_i be an abstract object in M. Then,*

$$O_i \in G_l \Longleftrightarrow O_i \in CG_l.$$

PROOF. By Corollary A.14. □

Theorem A.15 shows that for computation of local abstract objects of a method (i.e., objects that are not reachable from any non-local variable), using the connection graph of the method produces the same result as using the points-to graph of the method. Among the objects reachable from non-locals, we distinguish those reachable from globals or `Runnable` objects (*GlobalEscape*), and those not reachable from globals or `Runnable` objects (*ArgEscape*). Objects that are not reachable from globals or `Runnable` objects become part of the local subgraph for some method where they are no longer reachable from non-locals. Note that if an object node is marked *GlobalEscape* in any method, the corresponding nodes in its callees (and methods called transitively in those callees) are also marked *GlobalEscape*, as described in Section 4.5.

Objects reachable from globals or `Runnable` objects will be eventually propagated to `main()` (or root methods of the call graph), where the points-to graph will be identical to the connection graph because `main()` does not have any callsites invoking it. The following theorem formally states this.

THEOREM A.16. *Let O be an abstract object. There exists a path from a global variable or* `Runnable` *object to O in a points-to graph only if O is marked GlobalEscape in the connection graph.*

PROOF. Follows from the above discussion (in `main()` or root methods, $G^a_i \cong G^a_{cs} \oplus CG_i = CG_i$ because $G^a_{cs} = \emptyset$ for root methods). □

A.3 Reachability properties of escaping concrete objects

We now establish some properties of concrete objects that escape from their method or thread of creation.

LEMMA A.17. *Let O_c be a concrete object; let T be its thread of creation and M be its method of creation.*

—*O_c escapes T only if O_c is reachable from either (1) a static reference field or (2) a reference to a* `Runnable` *object.*

—*O_c escapes M only if O_c is reachable from either (1) a static reference field, (2) a formal parameter or return value of M, or (3) a reference to a* `Runnable` *object.*

PROOF. Follows directly from the Java language specification [Gosling et al. 1996]. Java allows an object O_c, created in thread T, to be accessed by another thread T' (where $T' \neq T$) only if it is either (1) reachable from a static field (no other form of "global" variables are supported in Java), or (2) reachable from the object representing thread T' or T.

Similarly, Java allows an object O_c, created in method M, to be accessed after M returns only if it is either (1) reachable from a static field, (2) reachable from a formal parameter or the return value (of an object reference type) of M, or (3) reachable from an object representing another thread that continues to run after M returns. □

A.4 Putting it together: Correctness of escape analysis

THEOREM A.18. *Let M be a method invocation in thread T, and A be an allocation site in M. At the end of our escape analysis, if the allocation site A is marked*

—*ArgEscape or NoEscape, then all the the concrete objects created at A are local to T.*

—*NoEscape, then all the concrete objects created at this site are local to M.*

PROOF. For the first part of the Theorem, consider a concrete object O_c that escapes its thread of creation. According to Lemma A.17, O_c must be reachable from p, where p is either a static field reference or a `Runnable` object reference. By Lemma A.2, there must exist an abstract object $O_a \in G^a$ that is reachable from p such that $AllocSite(O_c) \subseteq AllocSite(O_a)$. By Theorem A.16, the node corresponding to O_a in CG must be marked *GlobalEscape*. Thus, we have shown that a concrete object O_c can escape its thread of creation only if the corresponding abstract object O_a at its allocation site in CG is marked *GlobalEscape*. Therefore, if an abstract object is marked *NoEscape* or *ArgEscape* in CG, the corresponding concrete object cannot escape its thread of creation.

For the second part of the Theorem, consider a concrete object O_c that escapes its method of creation. According to Lemma A.17, O_c must be reachable from p, where p is a non-local variable with respect to M (i.e., p is either a formal parameter or return value of M, a static field reference, or a `Runnable` object reference). By Lemma A.2, there must exist an abstract object $O_a \in G^a$ that is reachable from p such that $AllocSite(O_c) \subseteq AllocSite(O_a)$. Thus, O_a is not part of G^a_l (the local points-to graph) for M. By Theorem A.15, the node corresponding to O_a in CG must not be part of the local subgraph of CG. Thus, we have shown that a concrete object O_c can escape its method of creation only if the corresponding abstract object O_a at its allocation site in CG is not marked *NoEscape*. In other words, if an abstract object is marked *NoEscape* in CG, the corresponding concrete object cannot escape its method of creation.

□

COROLLARY A.19. *If an allocation site A is marked*

—*NoEscape, then all the objects created at that site can be stack-allocated and the lock operations associated with that object can safely be eliminated.*

—*ArgEscape, then the locks associated with that object can safely be eliminated.*

B. TIME COMPLEXITY

In this section, we analyze the complexity of escape analysis. We will first discuss the complexity for intraprocedural case. We will assume that all deferred edges have been eliminated using the $ByPass(p)$ function. For intraprocedural analysis, we use the dataflow equations 1 and 2 for computing the connection graph. We can represent the connection graph as a set of pairs (x, y) such $x \rightarrow y$ is an edge in the connection graph. The fixed point can be obtained by iterating over the dataflow equations.

Let us assume that there are K nodes in the control flow graph. We can rewrite the equations 1 and 2 as follows:

$$\begin{aligned} C^{s_1} &= \cup_{p \in Pred(s_1)} f^{s_1}(C^p) \\ C^{s_2} &= \cup_{p \in Pred(s_2)} f^{s_2}(C^p) \\ &\vdots \\ C^{s_K} &= \cup_{p \in Pred(s_K)} f^{s_K}(C^p) \end{aligned}$$

Given that the transfer function is monotonic for escape analysis, we can compute the least fixed point to above equations using Kildall's iterative algorithm. Let A_{max} be the maximum size of the connection graph at any program point. Recall that an field access expression of the form $a.f1.f2...fl$ is broken into a series of one-level field access expressions of the $a.f$. During the construction of the connection graph, we essentially create one object node per field access. Since we are using 1-limited scheme for handling recursive structure, the maximum acyclic path length (an acylic connection is obtained by ignoring the back-edges in the connection graph) in the connection graph will limited by the maximum field length of the field expression. Let L be the maximum acyclic path-length in the connection graph.

Using the iterative scheme, we can compute the connection graph at a program point in $O(L \times A_{max})$. The reason for this is that at each iteration, we might introduce one object node in the connection graph by traversing the entire CG to check whether there is a change in the CG. Now since there are K nodes in the CFG, at each iteration we visit K nodes and update the connection graph at each of the nodes in the CFG. So at each iteration, the time complexity is $O(K \times L \times A_{max})$. At each iteration, we might add one edge in the connection graph, and since we add at most A_{max} edges, the number of iterations is bounded by $O(K \times A_{max})$. Therefore, the time complexity of intraprocedural analysis is $O(L \times A_{max}^2 \times K^2)$.

In the worst case, A_{max} can be $O(N^2)$. It quite reasonable to assume L to be constant. Also, K could be $O(N)$. Therefore the time complexity of intraprocedural analysis is $O(N^6)$. This complexity analysis is based on naive implementation of sets and the fixed point iteration. By using a worklist based strategy for computing the fixed point, we can reduce the complexity to $O(N^5)$. Finally, using finite difference technique, as proposed by Goyal in his thesis, which incrementally updates the connection graph, we can further reduce the complexity to $O(N^3)$ [Goyal 2000].

For the interprocedural case, our analysis essentially proceeds along the same lines, except that we use call graph and connection graph summary information at each call graph node. The transfer functions at each node is captured by the mapping function described in Section 4. The mapping function used to summarize the effects of a callee method has a complexity of $O(N^2)$. This does not increase the overall complexity of the analysis. Hence, the time complexity of interprocedural escape analysis is $O(N^6)$, where N is the program size. Using a clever scheme, we can reduce the complexity to $O(N^3)$ [Goyal 2000].