Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis

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

This article presents an *escape analysis* framework for Java to determine (1) if an object is not reachable after its method of creation returns, allowing the object to 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 nonescaping objects for methods and threads. 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 are eliminated in those 10 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 512 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

---

A preliminary version of this paper appeared in the *Proceedings of ACM OOPSLA '99*.
Authors' addresses: J.-D. Choi, M. Gupta, and V. C. Sreedhar, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: {jdchoi,sreedhar}@watson.ibm.com, mgupta@us.ibm.com; M. J. Serrano, Intel, 2200 Mission College Blvd., SC12-303, Santa Clara, CA 95054; email: mauricio.j.serrano@intel.com; S. P. Midkiff, School of Electrical and Computing Engineering, Purdue University, 465 Northwestern Ave., West Lafayette, IN 47907-2035; email: smidkiff@purdue.edu.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2003 ACM 0164-09251/03/1100-0876 $5.00

1. INTRODUCTION

The Java Programming Language [Gosling et al. 1996] provides a number of features that enhance programmer productivity but create challenges for the virtual machine implementation from a performance point of view. In particular, Java provides built-in support for automatic memory management and for concurrency. 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 and the lock 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. The term *escape analysis* has been used in the literature [Park and Goldberg 1992] for an analysis that determines 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). This analysis allows us to 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 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 contents of the stack frame are not garbage-collected). Also, by allocating objects on the local stack, we reduce the occasional synchronization that the heap allocator has to perform with other threads competing for memory chunks. Besides stack allocation, the knowledge about an object being local to a method can enable further optimizations. For example, more aggressive code reordering can be performed 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, an object access 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 low-level synchronization operations that ensure mutual exclusion on this object. Note that the Java memory model still requires that we refresh (flush) the Java local memory at `monitorenter` (`monitorexit`) statements in bytecode that are 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 briefly 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].

We introduce a framework for escape analysis based on a simple program abstraction called the *connection graph*. The connection graph 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 in either 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 [IBM Corporation 1997; Oliver et al. 2000], and have analyzed 10 medium to large benchmarks.

The main contributions of this paper are as follows:

— 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: 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 briefly describe an additional analysis that allows redundant memory 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 that is well suited for the purpose of escape analysis. It is different from the points-to graph for alias analysis whose major purpose is memory disambiguation. In the connection graph abstraction, we use the notion of *phantom nodes*, which allow us to summarize the effects of a callee procedure independent of the calling context.[1] This succinct summarization helps improve the overall speed of the algorithm.
— We present experimental results from an implementation of escape analysis in a Java compiler. We show that in 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 that more than 70% of objects are stack-allocatable in three programs). We are able to eliminate 11% to 92% of mutex lock operations in those ten programs. The overall performance improvements range from 2% to 23% on a 333-MHz IBM PowerPC workstation with 512-MB memory.

The rest of this paper is organized as follows. Section 2 presents our connection graph abstraction and formalizes the notion of the escape of an object. Sections 3 and 4, respectively, describe the intraprocedural and interprocedural

[1]Phantom nodes are similar to anonymous variables, hidden variables, ghost nodes, referred to in the pointer analysis literature [Landi and Ryder 1992; Choi et al. 1993; Emami et al. 1994; Wilson and Lam 1995; Rugina and Rinard 1999].

analyses that build the connection graph and identify the objects that do not escape their method or thread of creation. Section 5 elaborates on our handling of special Java features such as exceptions and object finalizers, and describes an extension (not incorporated in our current implementation) to detect cases in which the flush operations associated with Java locks can be eliminated. Section 6 describes the transformation and the run-time support for the stack allocation and lock elimination optimizations, and Section 7 presents experimental results. Section 8 discusses related work, and Section 9 presents conclusions. Appendix 9 derives the complexity of our algorithm. An extended version of this paper [Choi et al. 2002] presents a proof of correctness of our algorithm.

2. CONNECTION GRAPH REPRESENTATION FOR ESCAPE ANALYSIS

In Java, run-time instances of classes, called *run-time objects*, are created via `new` statements and are referenced by *object references*. A run-time object is composed of a collection of named fields. A field can be either an object reference or a nonreference (nonpointer) 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. Unless explicitly stated otherwise, we will use the term *object*(*s*) to mean *abstract object*(*s*) throughout this paper.

2.1 Connection Graph

A *connection graph* (CG) is a directed graph $\mathrm{CG} = (N_o \cup N_r, E_p \cup E_d \cup E_f)$, where

—N_o represents the set of objects.
—$N_r = N_l \cup N_a \cup N_f \cup N_g$ is the set of *reference nodes*, and
  —N_l represents the set of local reference variables (those locals and formals that are object references) in the program;
  —N_a represents the set of actuals, including return values, which are object references;
  —N_f represents the set of non-static fields that are object references (these are called *field reference nodes*); and
  —N_g represents the set of static fields, that is, *global variables* that are object references.
—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 $p \to q \in E_p$, then $p \in N_r$ 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 what is pointed to by q. If $p \to q \in E_d$, then $p, q \in N_r$.
—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 $p \to q \in E_f$, then $p \in N_o$ and $q \in N_f$.

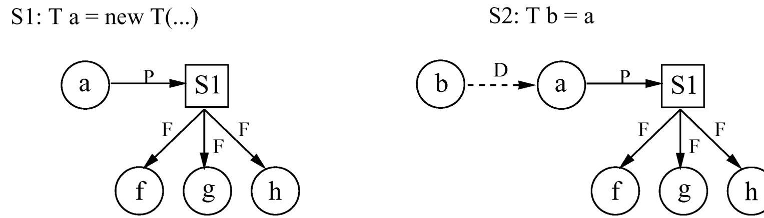

Fig. 1. A simple connection graph. Boxes indicate object nodes and circles indicate reference nodes (including field reference nodes). Solid edges to boxes indicate points-to edges, dashed edges indicate deferred edges, and solid edges from boxes to circles indicate field edges.

In our framework, we use a 1-limited naming scheme for creating objects [Chase et al. 1990]. We handle arrays in Java like regular objects, that is, we do not distinguish between different elements of an array.

Figure 1 illustrates an example of a connection graph. In figures, we represent each 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.

Let $PointsTo(p)$ denote the set of object nodes immediately pointed to by p. Given a CG and a reference node $p \in N_r$, we can compute $PointsTo(p)$ as follows: Traverse each outgoing path (consisting of zero or more deferred edges, followed by a points-to edge) from p until we reach an object node. The set of all such object nodes reachable from p will constitute $PointsTo(p)$.

We use deferred edges to model assignments that copy references from one variable to another. These edges defer computations during connection graph construction, and 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. 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 CG and adds edges in the set $\{r \xrightarrow{P} s | r \in R \textbf{ and } s \in S\} \cup \{r \xrightarrow{D} t | r \in R \textbf{ and } t \in T\}$ to the CG.

Deferred edges can improve the efficiency of the analysis by delaying, and thereby reducing the number of, graph updates. However, delaying these updates until the graphs are merged at control-flow join nodes can result in a CG path that consists of edges from mutually exclusive control-flow paths. This

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

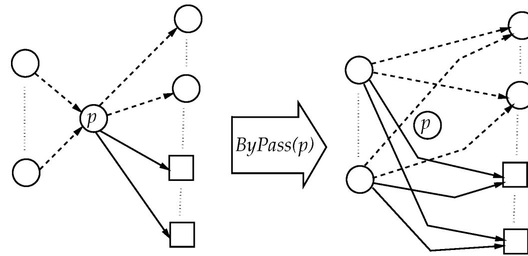

Fig. 2. Illustrating *ByPass(p)* function.

can result in a loss of information. The decision on when to apply the *ByPass* function should be made based on this tradeoff between the efficiency and the precision of the analysis. In our implementation, we choose effciency over precision, by not performing updates at control-flow join nodes.

2.2 Escape Property of Objects

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

Definition 2.1. 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 2.2. 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 model the concrete object graph using a CG. For each CG object node, we can identify an allocation site in the program where the concrete object instances are created for that CG node. Each node in the CG has an *escape state* associated with it. For marking escape states, we define an escape lattice consisting of three elements: *NoEscape*, *ArgEscape*, and *GlobalEscape*. 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*. 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*, this does not mean that all objects created *during* the execution of a thread will be marked *GlobalEscape*. The escape state of placeholder nodes representing actuals of a method (i.e., nodes in N_a) is initialized to *ArgEscape*. All other nodes are marked *NoEscape* initially.

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

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

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 mutex synchronization in accessing these concrete objects without violating Java semantics.

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

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

standard data flow equations:

$$C_o^s = f^s(C_i^s), \tag{1}$$
$$C_i^s = \wedge_{r \in Pred(s)} C_o^r, \tag{2}$$

where f^s denotes the *data flow transfer function* of node s, and the *meet* operation ($\wedge$) is a merge of connection graphs.

Given the bytecode simplification of Java programs, we identify four *basic statements* that have a nontrivial transfer function f^s: (1) $p = \text{new } \tau()$, (2) $p = q$, (3) $p.f = q$, (4) $p = q.f$. We present the transfer functions for each of these statements. Figure 3 illustrates the transfer functions for each of the basic statements for flow-sensitive analysis.

(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. The difference is that we perform *strong updates* with flow-sensitive analysis, but perform *weak updates* with flow-insensitive analysis.

(2) $p = q$. As in the previous case, for flow-sensitive analysis, we first apply *ByPass(p)*, and then add the edge $p \xrightarrow{D} q$: we perform strong updates. Again, for flow-insensitive analysis we ignore *ByPass(p)* but add the edge $p \xrightarrow{D} q$: we perform weak updates.

(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 is reachable from a formal parameter). We conservatively assume the second possibility (if $U = \emptyset$), 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. We also use a 1-limited scheme for creating phantom nodes.

Now let $V = \{v | u \xrightarrow{F} v \textbf{ and } u \in U \textbf{ and } fid(v) = f\}$, where $fid(v)$ is the field id of the field node v. For each $u \in U$, if the corresponding field reference node v does not exist, 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.

An assignment to a static field of a class is a special case of this transfer function: all the objects pointed to by q of the right-hand side will become *GlobalEscape*.

(4) $p = q.f$. Let $U = PointsTo(q)$, $V = \{v | u \xrightarrow{F} v \textbf{ and } u \in U \textbf{ and } fid(v) = f\}$. As in the previous case, if U is empty, we create a phantom node and add it to U. Also, we create a field reference node v, if necessary, for each node $u \in U$, 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 add the edges in $\{p \xrightarrow{D} v | v \in V\}$ to the connection graph.

This transfer function also applies when f is a static field of class q.

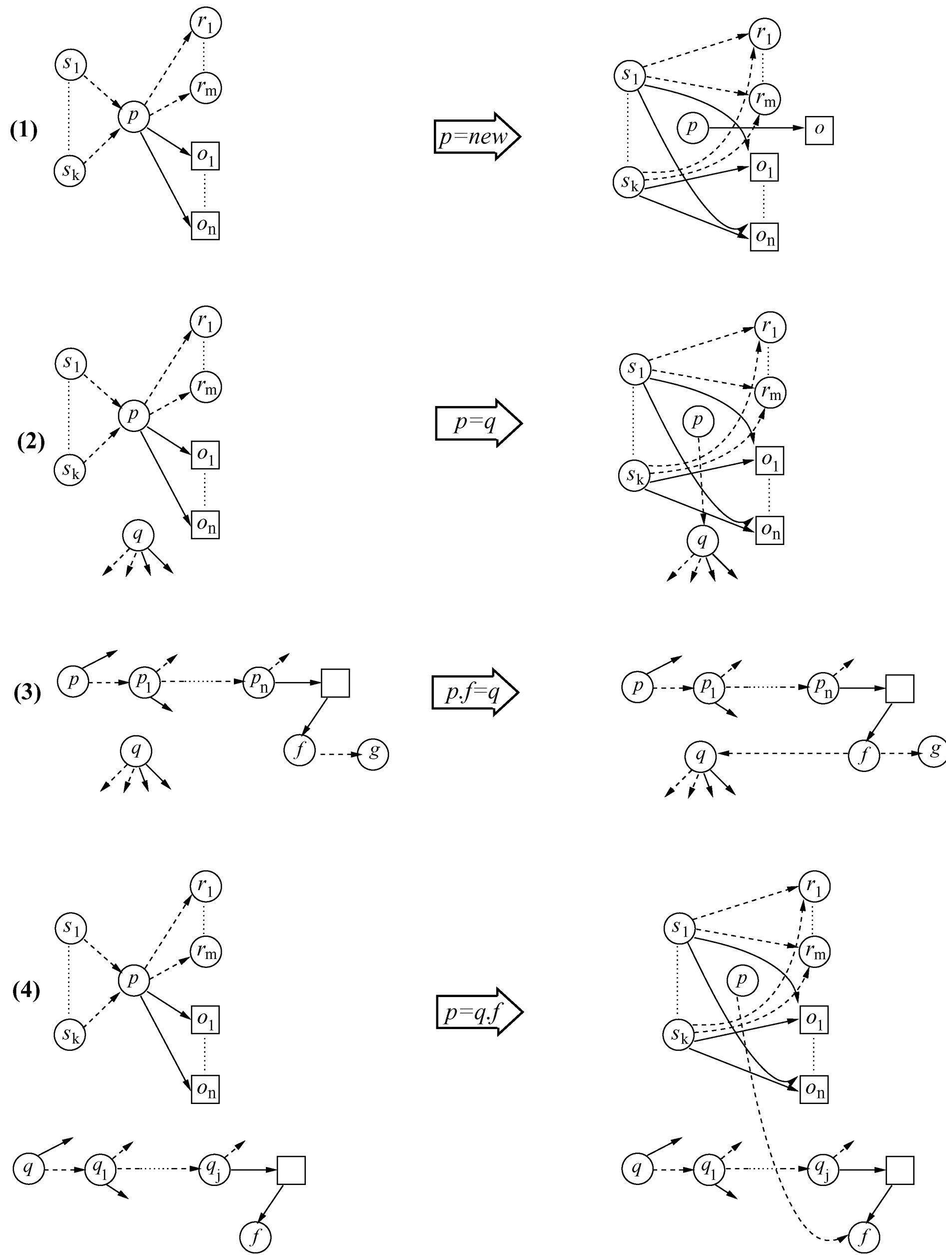

Fig. 3. Flow-sensitive transfer functions for basic statements.

We define the 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. If N_1 and N_2 have common nodes, that is, 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

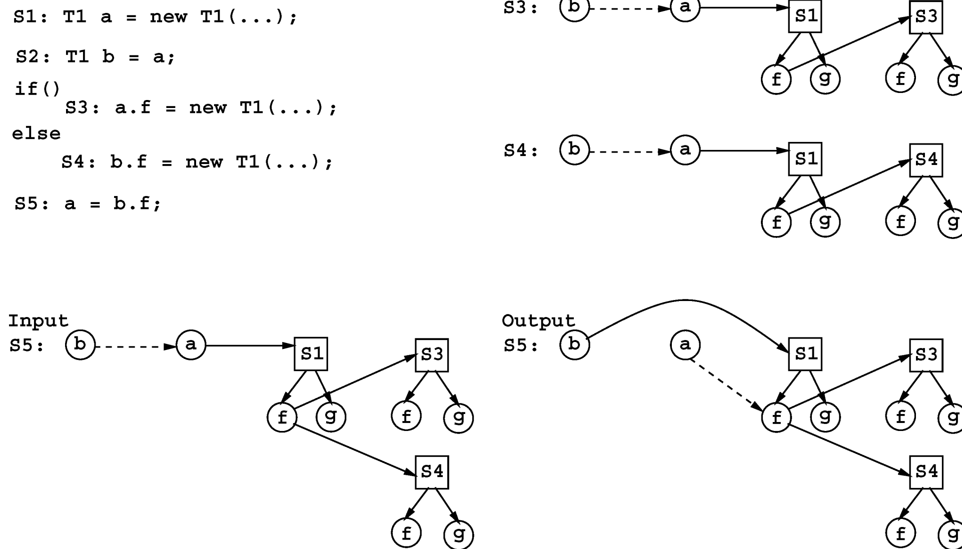

Fig. 4. An example illustrating connection graph computation. The connection graphs at S1 and S2 are not shown.

formally,

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

Let n_3 be a node in C_3. The escape state of $n_3 \in C_3$ is

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

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 10) on the number of iterations. If convergence is not reached, a `bottom` solution, in which every object node is marked *GlobalEscape*, is assumed for that method. In practice, we did not encounter this situation.

Figure 4 illustrates an example showing the connection graphs at various program points computed using the analysis described in this section.[4] The connection graphs labeled "`S3:`" and "`S4:`" are the graphs right after the corresponding statements.

For simplicity, we have referred to different connection graphs at different program points. However, in our implementation, we maintain a single connection graph for each method, which is updated incrementally. In the flow-insensitive version of our analysis, all connection graph updates can be made *in place*. In the flow-sensitive version, we only kill local variables, and add a

[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;`.

node at an update of a local variable. We add this node in a 1-limited manner. At the entry and exit of each basic block, we keep track of the mapping from local variables to nodes representing those variables at that point. At a control flow join point, while merging two local nodes n_1 and n_2 for a local variable, we create a new node n_3 if neither n_1 nor n_2 has connections that subsume the connections of the other node. Again, we use a 1-limited scheme in creating such a node, so as to avoid creating multiple nodes while iterating over the data flow solution for a loop.

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 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 in insensitive to 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 is sufficient for this phase. In order to handle cycles (due to recursion), we iterate over the nodes in strongly connected components of the 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—that is, all nodes in the set N_a (for actual arguments and return value) for methods involved in a nonconverging strongly connected component are marked *GlobalEscape*. This phase is sufficient to identify stack-allocatable objects, which are also thread-local.

In the presence of a method whose body cannot be analyzed, we mark objects reachable from any of the actual arguments passed to such methods as *GlobalEscape*. Examples of these methods are native methods not implemented in Java or methods of dynamically loaded classes whose body cannot be determined during analysis. This is necessary because these methods can pass an object reachable from its parameter to a native method, which in turn can make the object reachable by another native method of a different thread. Since Java allows a native method to invoke synchronization operations on a Java object that it can access through Java Native Interface (JNI), any object reachable from an actual argument passed to an unknown Java (or non-Java) method should be considered as escaping the current thread.

In order to identify additional thread-local objects, that is, 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,

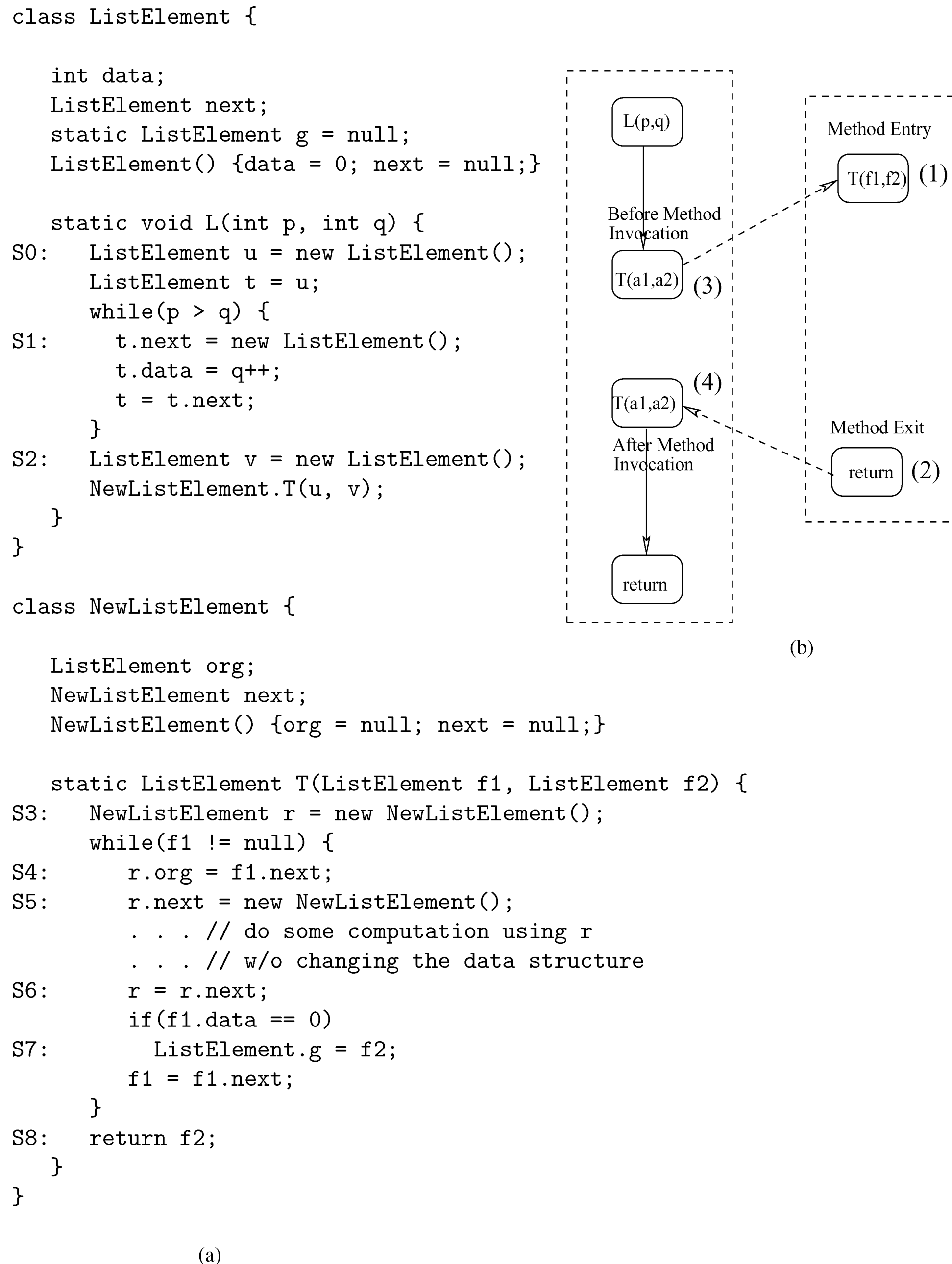

Fig. 5. An example program for illustrating interprocedural analysis and its call graph.

is performed in a separate top-down pass over the PCG, which is described in Section 4.5.

We will use the Java example shown in Figure 5 to illustrate our interprocedural framework. In this example, method $L()$ constructs a linked list and

method $T()$ constructs a tree-like structure. Figure 5(b) shows the caller-callee relation for the example program, shown in Figure 5(a). In Figure 5(b), we identify the four points of interest to interprocedural analysis, which are discussed in the following subsections. Figure 6 shows the connection graphs built at various points of the program in Figure 5(a). These connection graphs are a conservative representation of the data structure built by the program during execution.

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 an extension to identify more thread-local objects. Section 4.6 presents a comparison between CG and points-to graph.

4.1 The 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 argument 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 $EscapeState[a_i] = ArgEscape$. Figure 6(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 The 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). The round node r in Figure 6(d) represents the phantom return node. 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 first use the *ByPass* function (defined in Section 2) to eliminate all the deferred edges in the CG, except for those pointing to a *terminal* node, which is a variable or field node without any oytgoing (points-to) edges. (Since terminal nodes do not have outgoing edges, the *ByPass* function cannot be applied to them.) In order to remove the deferred edges to terminal nodes, we create a phantom node for each terminal node, and insert a points-to edge from the

[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.

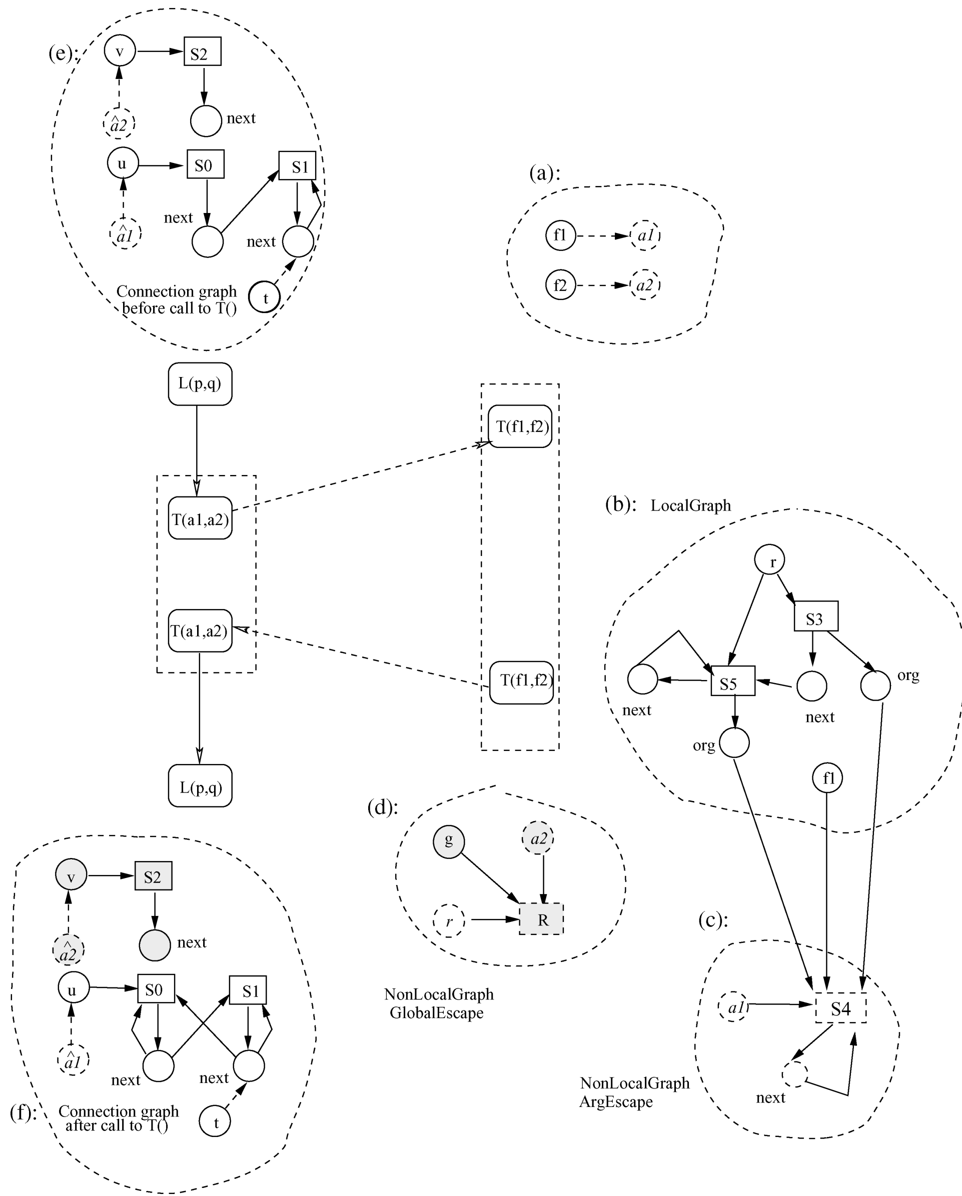

Fig. 6. Connection graphs at various points in the call graph. Nodes that escape globally are shadowed. (a) shows CG at method entry in the callee. (b), (c), and (d) show the different subgraphs (respectively, *LocalGraph*, *ArgEscape NonLocalGraph*, and *GlobalEscape NonLocalGraph*) of CG at method exit in the callee. (e) shows CG before the method call in the caller. (f) shows CG after the method call in the caller.

terminal node to the phantom node. We then apply the *ByPass* function to each of the terminal nodes, which removes the deferred edges to the terminal nodes. However, we keep the points-to edges from the terminal nodes to the phantom nodes.

As an example, Figure 6(b), (c), and (d) show the connection graph at the exit of method `T()`. The (rectangular) phantom object node R in Figure 6(d) represents the phantom object pointed-to by `f2` when `f2` is returned by method `T`. After the first application of the *ByPass* function, terminal node `a2` had three incoming deferred edges, one from each of `f2` (in Figure 6(a)), `g` (created at `S7`), and `r` (created at `S8`). After creating the phantom node R and inserting a points-to edge from `a2` to R, deferred edges to `a2` were removed by applying the *ByPass* function to `a2`. (In the example code, the return value from method `T` is ignored by method `L`, the caller of `T`.)

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 6(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 *nonlocal 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 nonlocal subgraph, but not vice versa. All objects in the local subgraph that are created in the current method are marked stack-allocatable. The nonlocal 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.

Getting back to the running example, Figure 6(b), (c), and (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 6(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 target phantom object of the phantom reference node *a2* become *GlobalEscape*, as shown in Figure 6(d). Figure 6(d) already shows the effect of applying the *ByPass* function. Before the application, the graph corresponding to Figure 6(d) has four reference nodes and deferred edges between them: reference nodes $g, r, f2$, and $a2$, with deferred edges from g to $f2$ due to `S7`, from r to $f2$ due to `S8`, and from $f2$ to $a2$ inherited from Figure 6(a). The summary graph for method `T()` will consist of the nonlocal subgraphs shown in Figure 6(c) and (d).

4.3 The Connection Graph Immediately Before a Method Invocation

At a method invocation site, passing of each parameter is handled as an assignment to an actual argument $\hat{a}_i$ 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 arguments to *foo*(). We model the call as follows: $\hat{a}_1 = u_1; \hat{a}_2 = u_2; \ldots; foo(\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n)$. Each $\hat{a}_i$ at the call site will be matched with the phantom reference node a_i of the callee method. In Figure 6(e), two nodes, $\hat{a1}$ and $\hat{a2}$, are created with deferred edges pointing to the first and the second actual arguments to the call, `u` and `v`, respectively.

4.4 The 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). Two 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 (representing actual arguments and return value) of the caller's CG, and a_i's 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*. If the callee is a virtual method, we update the CG at the caller with the summary information from each possible target at that site, effectively merging the graphs of all target methods into the caller's CG.

4.4.1 *Updating Caller Nodes.* Figure 7 describes how we map the nodes in the callee's CG with the nodes in the caller's CG. This mapping of nodes from the callee CG to the 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* ($\longmapsto$) recursively among objects belonging to a callee CG and a caller CG as follows:

Definition 4.1

(1) $a_i \longmapsto \hat{a}_i$.
(2) $O_p \in PointsTo(p) \longmapsto \hat{O}_q \in PointsTo(q)$, 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 7, *MapsToObj*(n) denotes the set of objects that n can be mapped to using the *MapsTo* relation discussed above. In the figure, we use f and $\hat{f}$ to denote a callee node (field) and a caller node (field), respectively. The algorithm starts with a_i and $\hat{a}_i$ as the original nodes 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.

```
UpdateCallerNodes()
{
1:  foreach a_i, â_i actual argument pair do
2:     UpdateNodes (a_i, {â_i});
3:  endfor
}

UpdateNodes( f: node; // actual or field
               mappedFields: set of nodes) // actual or field
// mappedFields is the set of MapsTo field nodes of f
{
4:  foreach object node n_o ∈ PointsTo(f) do
5:     foreach f̂ ∈ mappedFields do
6:        if PointsTo(f̂) = ∅ then
7:           CreateTargetNode( f̂ ); // create/insert a new node as the target of f̂
8:        endif
9:        foreach n̂_o ∈ PointsTo(f̂) 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 g such that n_o -F-> g do
14:                tmpMappedFields = {ĝ | n̂_o -F-> ĝ and fid(g) = fid(ĝ)};
15:                UpdateNodes( g, tmpMappedFields);
16:             endfor
17:          endif
18:       endfor
19:    endfor
20: endfor
}
```

Fig. 7. Algorithm to update the caller's connection graph nodes.

The main body of procedure UpdateNodes is applied to all the callee object nodes pointed to by the callee field node f (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 $\hat{f}$, which is itself a *MapsTo* field node of callee node f (i.e., $\hat{f} \in mappedFields$). In Statement 7 caller object node $\hat{n}_o$ and its field nodes are created with an escape state of *NoEscape* if no *MapsTo* caller object node exists (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., g), the set of *MapsTo* field nodes of the caller (i.e., *tmpMappedFields*). It then recursively invokes UpdateNodes, passing g and *tmpMappedFields* as the new parameters (Statement 15).

4.4.2 *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 7 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 $\hat{p} \stackrel{F}{\rightarrow} \hat{f}_p \stackrel{P}{\rightarrow} \hat{q}$ by inserting a points-to edge $\hat{f}_p \stackrel{P}{\rightarrow} \hat{q}$ for each field node $\hat{f}_p$ of $\hat{p}$ such that $fid(f_p) = fid(\hat{f}_p)$.

4.4.3 *Example.* Consider the summary nonlocal subgraphs shown in Figure 6(c) and (d). First, all nodes that are reachable from global variable g are marked *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 construct the connection shown in Figure 6(f) that is right after the invocation site of `T()`. We first mark the phantom reference node *a1* of the callee (in Figure 6(c)) and the phantom node $\hat{a1}$ of the caller (in Figure 6(f)) as the initial nodes, that is, a_i and $\{\hat{a}_i\}$ at Statement 2 in Figure 7. Then we map the phantom node `S4`, pointed to by *a1*, to `S0`, pointed to by $\hat{a1}$. The cycle in the nonlocal 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, that is, we create at most one phantom node at a statement. Now since *a2* is marked *GlobalEscape*, all the nodes of the caller reachable from $\hat{a2}$ will also be marked as *GlobalEscape*.

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. We perform this propagation of *GlobalEscape* state in a separate top-down traversal over the PCG. Cycles in the PCG are handled by iterating until the solution converges (we observed rapid convergence in practice). 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, unlike the work presented in Ruf [2000].) Callee's nodes which correspond to caller's nodes are identified using the inverse of the *MapsTo* relation described in Section 4.4. 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 Connection Graph Versus Points-To Graph

The connection graph has some similarities to, but also a few important differences from the *points-to* graph representation that has been proposed in the

literature for pointer analysis [Chase et al. 1990; Sagiv et al. 1998]. 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 Chase et al. [1990], Choi et al. [1993], and 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.*

The connection graph lacks the uniqueness property since a concrete object can be mapped to multiple abstract objects. Even so, the connection graph can still be used for computing certain static properties of the program such as whether or not objects escape. More detailed comparisons of the connection graph and the points-to graph can be found in Choi et al. [2002].

5. HANDLING JAVA-SPECIFIC FEATURES

In this section, we show how we handle Java-specific features related to exceptions, object finalization, and synchronization side-effects.

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 statements 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 (from Rules 1, 2, and 4 in Figure 3) that we "kill" only local reference variables of a method in a flow-sensitive analysis. Therefore, we only need to worry about those variables. Of the local variables updated 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 at an assignment inside that block, 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 nonlocal 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 have a correct 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 a class overriding the finalizer as *GlobalEscape*.

5.3 Synchronization Side-Effects

Java synchronization not only provides a mechanism to enforce mutual exclusion, but it also has side-effects related to the state of shared variables accessed

within the mutex block. In particular, the semantics of the *acquire* phase of the synchronization operation (`monitorenter`) 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 instruction (*isync* in the PowerPC architecture). The semantics of the *release* phase of the synchronization operation (`monitorexit`) are to (1) make sure that all writes to memory of global shared values are completed, and (2) release the lock. Enforcement of the first requirement involves executing a storage synchronization instruction, for example, the Sparc *fence* or PowerPC *sync* instruction. *Sync* operations can be expensive, particularly on multiprocessor systems, since each memory subsystem must acknowledge the completion of all writes initiated by the processor executing the *sync*. Even if the lock operation is shown to be unnecessary by escape analysis, it may still be necessary to execute the *sync* instruction to ensure the completion of writes to global memory operations. We now outline an unimplemented algorithm for determining when the *sync*, as well as the lock release, can be removed.

Before considering how to determine when a *sync* operation can be removed, 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 (including paths resulting from exceptions) without an intervening *sync* operation, and (2) there exists a corresponding r operation for the w operation. These r operations can be found by examining the conflict edges adjacent to w in the CSSA graph of Lee et al. [1999]. Intraprocedurally, the data flow analysis is straightforward, and follows from the requirements above. Using a simple bitvector analysis, it can be determined which *sync* instructions s can be reached by a global write, with no intervening *sync* instructions. The *sync* instructions s must be kept—all others can be removed. Details of the analysis can be found in Choi et al. [2002].

6. 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 [IBM Corporation 1997; Oliver et al. 2000]: (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 escape analysis, and the run-time support to implement these optimizations.

6.1 Transformation

At the end of the analysis described in Section 4, we mark each `new` site in the program as follows: (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.

6.2 Run-Time Support

We allocate objects on the stack by calling the native `alloca` routine in HPCJ's AIX backend. Each invocation of `alloca` grows the current stack frame at run time. 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 is possible to allocate a fixed piece of storage on the stack frame for that `new` statement. Our current 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 its contents are not garbage-collected.

A secondary benefit of stack allocation is the elimination of occasional synchronization for allocation of objects from the thread-common heap. Most heap allocations by the HPCJ runtime system (like many other Java Virtual Machines) are satisfied by allocations from a thread-local heap, and do not require any synchronization. For allocating a large object, or when the local heap space is exhausted, the given thread needs to allocate from thread-common heap space, which requires a relatively heavy-weight synchronization. Allocating objects on the stack reduces the requirement for allocations from the thread-common heap space.

Elimination of synchronization operations requires run-time support at two places: allocation sites of objects, that is, `new` sites; and use sites of objects as synchronization targets, that is, `synchronized` methods or statements. Synchronized methods and statements are supported using `monitorenter` and `monitorexit` atomic operations, whose implementation 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 that indicates whether or not the object is thread-local. At object use sites, we modify 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 nonatomic 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 refresh and flush the local memory at synchronization points to ensure that locally cached copies of global variables are read from, and written to, global variables as required by the 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.

Table I. Benchmarks Used in Our Experiments

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 II. Benchmark Characteristics

	Number of objects allocated		Size of objects in bytes allocated		Total number of locks	
Program	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

7. 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 data flow information. 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 the 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

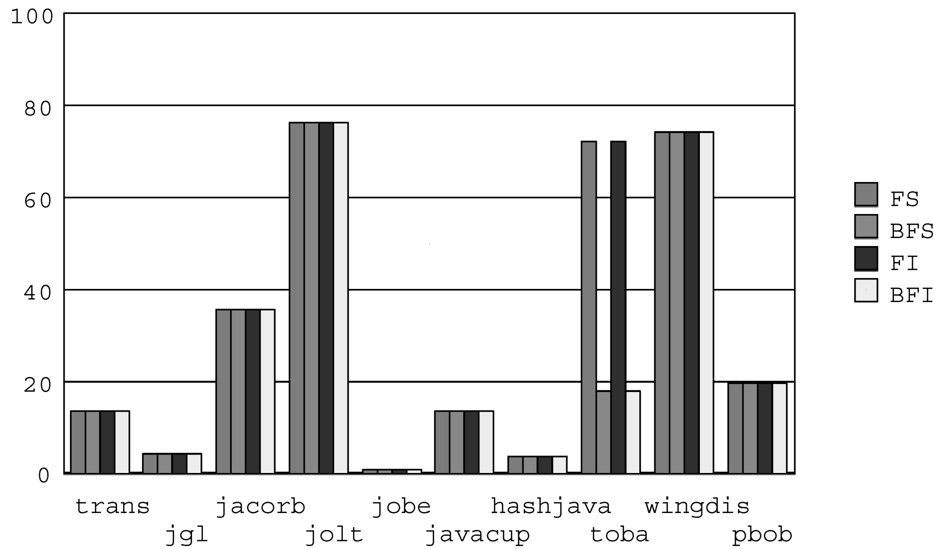

Fig. 8. Percentage of user code objects allocated on the stack.

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 the 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 7.1 discusses results for stack allocation of objects. Section 7.2 discusses results for synchronization elimination. Section 7.3 discusses the actual execution time improvements due to these two optimizations.

7.1 Stack Allocation

Figure 8 shows the percentage of user objects that we allocate on the stack, and Figure 9 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). These benchmarks tend to create many objects of the `StringBuffer` class, which are especially amenable to our analysis for stack allocatability. 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 fields 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

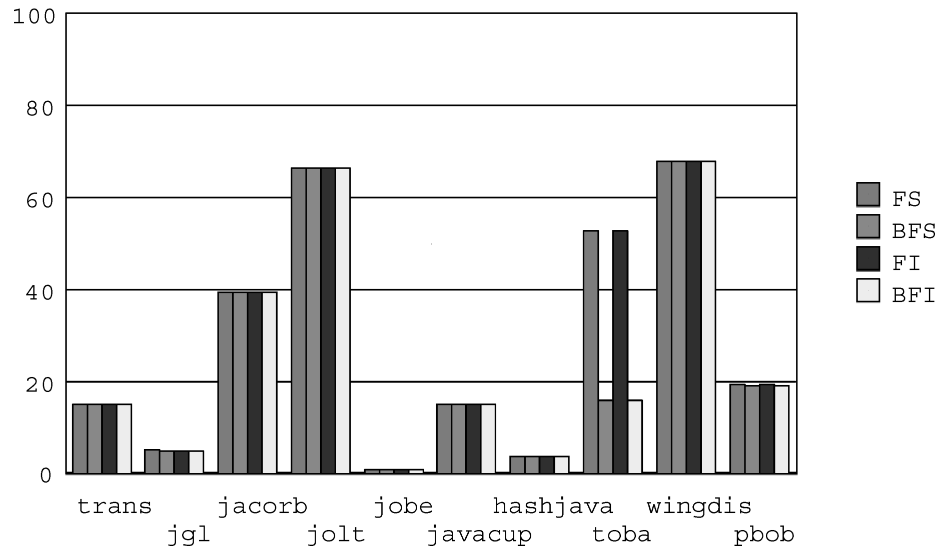

Fig. 9. Percentage of user code object space allocated on the stack.

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. This is due to multiple fields with different escape states being mapped to the same node, leading to conservative analysis results. We tried different functions for mapping the fields to the given bounded number of nodes, and found that it did not make much difference to the quality of results.

7.2 Lock Elimination

For lock elimination, we collected two sets of data for different variants of the analysis. We measured 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 10 shows the percentage of user objects that are determined to be local to a thread, and Figure 11 shows the percentage of lock operations that are removed during execution based on the identification of thread-local objects. It can be seen that our most precise analysis version finds many 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, the percentage of thread-local objects (≈50%) is higher than the percentage of locks removed (≈15%), suggesting that relatively few thread-local objects are actually involved in synchronization.

For wingdis, we found a large percentage of objects are thread-local (≈75%), and were able to remove ≈91% of synchronization operations. Notice that jobe has very few objects, less than 1%, identified as thread-local. However, there are a relatively large number of synchronization operations performed on them, leading to an opportunity for eliminating a higher percentage of

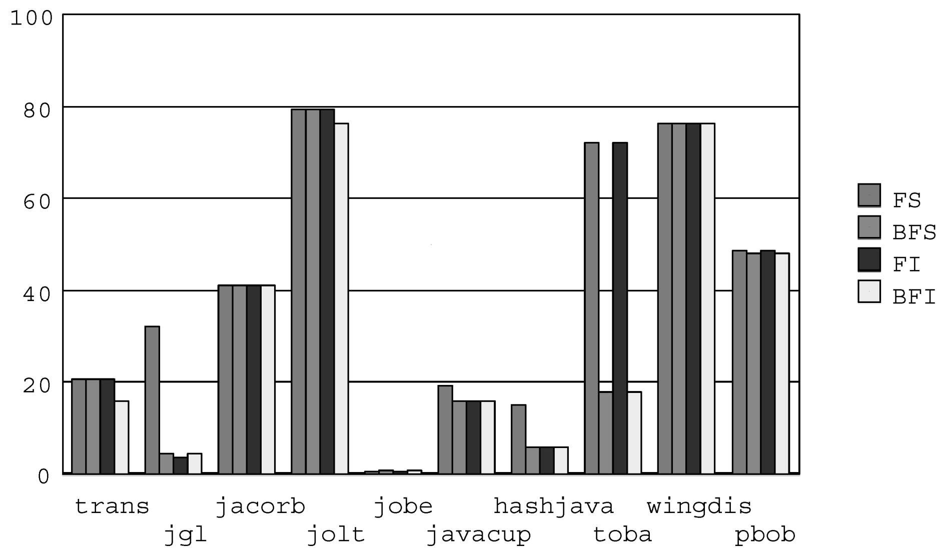

Fig. 10. Percentage of thread-local objects in user code.

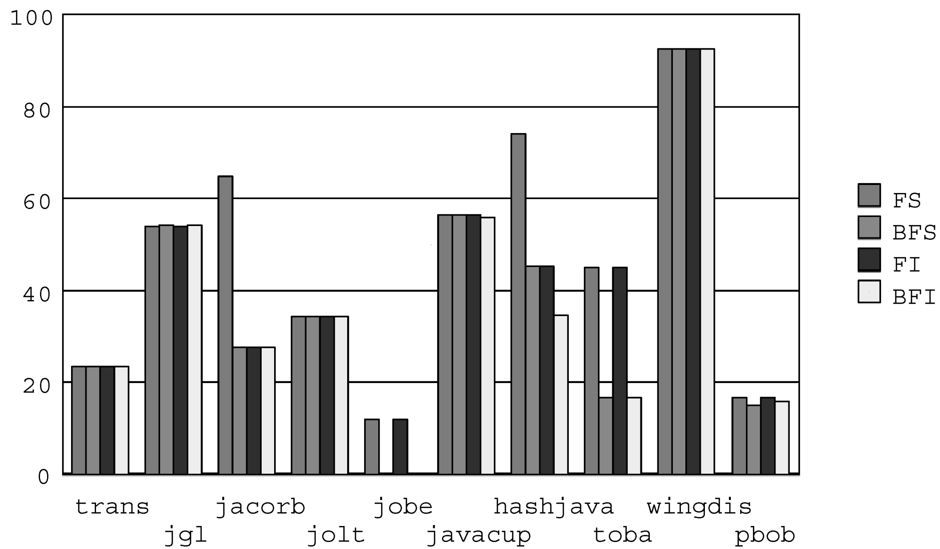

Fig. 11. Percentage of locks removed on objects in user code.

synchronization operations. The versions of our analysis using an 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 are 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

Table III. Improvements in Execution Time

Program	Execution time (s)	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

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.

7.3 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 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. The `pbob` benchmark was run on a four-way PowerPC SMP machine. All of these benchmarks, except for `jgl` and `pbob`, allocate less than 100 MB of heap space cumulatively, and had very little garbage collection activity. Hence, the performance gains come mainly from synchronization elimination rather than from stack allocation.

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. The `jgl` benchmark had a significant percentage of thread-local objects, and a correspondingly high percentage of locks removed, which contributed to its good performance. The `wingdis` and `toba` benchmarks shared these characteristics. The synchronization operations represent a smaller fraction of execution time for `wingdis` than for `jgl`—both programs have comparable execution times, and as Table II shows, the number of lock operations in `wingdis` is an order of magnitude smaller than the number of lock operations in `jgl`. This explains the smaller percentage performance improvement for `wingdis` compared to `jgl` in spite of the larger fraction of synchronization operations eliminated.

Table III 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 5.3. This data suggests that the actual realizable gains may be marginal. 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.

8. 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 [1992] introduced the term *escape analysis* 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]. Tofte and Talpin [1994] proposed 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 [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 allocate objects that do not escape a method and are created in the method. In some cases, it is possible to preallocate objects that escape a method [Gay and Aiken 1998; Gay and Steensgaard 2000], a technique that is akin to region allocation.

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

The current article is an extended version of Choi et al. [1999], and includes a more detailed discussion of various optimizations and analysis. There have been a number of parallel efforts on escape analysis for Java [Choi et al. 1999; Bodga and Hölzle 1999; Aldrich et al. 1999; Whaley and Rinard 1999; Blanchet 1999; Reid et al. 1999; Gay and Steensgaard 2000; Ruf 2000]. These efforts have provided different (partial) solutions to the (undecidable) problem of eliminating unnecessary lock operations. In particular, these efforts represent different tradeoffs between precision and complexity of analysis, which affects the balance between the quality of results on the one hand and compilation time and scalability of the analysis to large programs on the other. Some of these approaches violate the semantics of the Java memory model when attempting to eliminate unnecessary lock operations. We will now provide a detailed comparison of these approaches, so that the reader can make a better judgment about which technique to use in a given situation.

Bodga and Hölzle [1999] used set constraints for computing thread-local objects. Their system is a bytecode translator, and uses replication of execution

paths as the means for eliminating unnecessary synchronization. After replication, they converted synchronized methods that access only thread-local objects into nonsynchronized 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 5.3). They also summarized the effect of native methods, although manually. Using the summary information, they improved the precision of their analysis. Our approach can be easily extended to include native method analysis. We now discuss some more fundamental advantages of Bodga and Hölzle's [1997] approach relative to ours, as well as disadvantages. First, their approach uses a byte code translator and simply converts synchronized methods to unsynchronized methods. For single-threaded applications, this is an effective approach and does not violate Java semantics (note that even in single threaded applications, Java library methods could be synchronized). Second, their approach is less conservative in treating `new` objects. We create a single object node for a `new` statement regardless of the calling context. So if an object escapes its thread of creation in one context, it will be marked as escaping the thread of creation in all contexts. Bodga and Hölzle [1999] replicated and specialized objects depending on the calling context. Replications increases code size, but leads to improvements in some applications.

However, Bodga and Hölzle's [1999] basic constraint based approach is less precise than our approach. For the common benchmark application, `javacup`, Bodga and Hölzle were able to convert about 33% of the synchronized methods, while we were able to eliminate about 56% of the lock operations.

Whaley and Rinard's [1999] approach has many similarities to our approach, and represents an improvement over ours along two fronts: their program abstraction can be used to do both memory disambiguation and perform escape analysis. Similar to our connection graph, Whaley and Rinard's points-to escape graph can be summarized independent of the calling context (they used the term *compositional* for describing this property). A second improvement is that they performed strong updates on field variables. To allow strong updates, they computed must-alias information, which they could do since their program abstraction captures memory disambiguation. Furthermore, their implementation handles both user and library code, whereas ours modifies object allocation only in user code. There are two benchmarks that are common between the two papers: `javacup` and `pbob`. For `pbob`, Whaley and Rinard [1999] were able to allocate about 27% of the objects on stack whereas we allocate about 20% of objects on stack. They were able to eliminate about 24% of the lock operations, while we eliminate about 17% of locks. Similarly, they obtained better results on the `javacup` benchmark. In a more recent paper, Vivien and Rinard [2001] used a demand-driven approach to speed up the analysis.

Aldrich et al. [1999] described 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. They also removed synchronization operations, which can break Java semantics. They claimed 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 [1999] used 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. The escaping part of an object was represented by the height of its type. He proposed a two-phase flow-insensitive analysis, consisting of a backward and a forward phase, for computing escape information. He used 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 tested at runtime whether o was on the stack—if it was, the synchronization was 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. Blanchet's algorithm is inherently less precise, but faster than our algorithm.

Ruf's [2000] analysis computed threads that access globally escaping objects. If only one thread accesses such globally escaping objects, synchronization operations can be eliminated on them. Thus, Ruf's analysis was less conservative than ours when treating globals. Ruf also specialized procedures, when necessary, to perform more aggressive synchronization elimination. Ruf's approach seems to be the most effective in terms of eliminating unnecessary locks. He was able to eliminate almost all of the lock operations in many applications, including `javacup`, because of the way he modeled the objects. Rather than reasoning in terms of escaping a thread, Ruf modeled whether an object can potentially be synchronized by more than one thread during the object's lifetime. If so, the lock operations on that object cannot be eliminated.

To reduce the size of finite-state models of concurrent Java programs, Corbett [1998] uses a technique called *virtual coarsening*. 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 used a simple intraprocedural pointer analysis, after method inlining, to identify the heap objects that were local to a thread, and also to identify the variables that were guarded by various locks.

Pugh [1999] described some problems with the semantics of the Java memory model, and is leading an effort to revise the memory model [JavaMemoryModel 2002; Manson and Pugh 2001]. All memory flush operations associated with thread-local objects may be eliminated without further analysis (as described in Section 5.3) under one of the proposals being considered for the revised Java memory model [JavaMemoryModel 2002].

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; Sagiv et al. 1998; Ghiya and Hendren 1998; Chatterjee et al. 1999]. Conventional representations for pointer analysis, including the alias graph and points-to graph, have been 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 of memoization, we do lose some precision in the case where the resolution to a function pointer can be made more precise by reanalyzing the procedure. Recently Cheng and Hwu [2000] and Liang and Harrold [1999] improved the space requirement of alias analysis essentially by summarizing and mapping information across method boundaries.

The connection graph is a simpler abstraction than the alias or points-to graph. For identifying stack-allocatable objects, we can summarize the connection graph on a per procedure basis regardless of the incoming calling context. Hence, in this context, connection graph analysis is amenable to elimination-based data flow analysis [Marlowe 1989], that is, one can construct a closure operation that essentially summarizes the effects of a method. Also, 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 into the caller connection graph is simpler. Fähndrich et al. [2000] also observed the above property and used directional information to speed-up queries on data flow information.

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

9. CONCLUSIONS

In this paper, we have presented a new interprocedural algorithm for escape analysis. Apart from using escape analysis for the 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 dynamic mutex lock operations in our benchmarks, eliminating more than 50% of the mutex 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.

APPENDIX: TIME COMPLEXITY

In this appendix, 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 data flow Equations (1) and (2) for computing the connection graph. We can represent the connection graph as a set of pairs (x, y) such $x \to y$ is an edge in the connection graph. The fixed point can be obtained by iterating over the data flow equations.

Let us assume that there are K nodes in the control flow graph. We can rewrite Equations (1) and (2) (Section 3) 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 a field access expression of the form $a.f1.f2 \cdots fl$ 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 [2000], which incrementally updates the connection graph, we can further reduce the complexity to $O(N^3)$.

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 function 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 Goyal's [2000] finite-differencing technique we can reduce the complexity to $O(N^3)$.

ACKNOWLEDGMENTS

We would like to thank David Bacon, Michael Burke, Mike Hind, Deepak Goyal, Ganesan Ramalingam, Vivek Sarkar, Ven Seshadri, Marc Snir, and Harini Srinivasan for useful technical discussions. We also thank the 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* (Venice, Italy).

BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. 1998. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Montreal, P.Q. (Canada).

BIRKEDAL, L., TOFTE, M., AND VEJLSTRUP, M. 1996. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*.

BLANCHET, B. 1998. Escape analysis: Correctness, proof, implementation and experimental results. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA). 25–37.

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

BODGA, J. AND HÖLZLE, U. 1999. Removing unnecessay synchronization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, CO).

BURKE, M., CARINI, P., CHOI, J.-D., AND HIND, M. 1995. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.). Lecture Notes in Computer Science, Vol, 892. Springer-Verlag, Berlin, Germany, 234–250. Extended version published in September 1994 as Res. rep. RC 19546, IBM T. J. Watson Research Center, Yorktown Heights, NY.

CHAMBERS, C., PECHTCHANSKI, I., SARKAR, V., SERRANO, M. J., AND SRINIVASAN, H. 1999. Dependence analysis for Java. In *Proceedings of the 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 *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*. *SIGPLAN Not. 25*, 6, 296–310.

CHATTERJEE, R., RYDER, B. G., AND LANDI, W. A. 1999. Relevant context inference. In *Proceedings of the 26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*.

CHENG, B.-C. AND HWU, W.-M. 2000. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM*

SIGACT-SIGPLAN Symposium on the Principles of Programming Languages. 232–245.

CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. 1999. Escape analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, CO).

CHOI, J.-D., GUPTA, M., SERRANO, M. J., SREEDHAR, V. C., AND MIDKIFF, S. 2002. Stack allocation and synchronization optimizations for Java using escape analysis. Res. rep. RC22340. IBM T. J. Watson Research Center, Yorktown Heights, NY.

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, New York, NY.

DEUTSCH, A. 1997. On the complexity of escape analysis. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA). 358–371.

DINIZ, P. AND RINARD, M. 1997. Synchronization transformations for parallel computing. In *Proceedings of the 9th 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*. 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*. 253–263.

GAY, D. AND AIKEN, A. 1998. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, P.Q., Canada).

GAY, D. AND STEENSGAARD, B. 2000. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the 2000 International Conference on Compiler Construction*. 82–93.

GHIYA, R. AND HENDREN, L. J. 1996. Connection analysis: A practical interprocedural heap analysis for C. *Int. J. Parallel Program. 24*, 6, 547–578.

GHIYA, R. AND HENDREN, L. J. 1998. Putting pointer analysis to work. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA). 121–133.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java$^{(TM)}$ Language Specification*. Addison-Wesley, Reading, MA.

GOYAL, D. 2000. A language-theoretic approach to algorithms. Ph.D. dissertation, New York University, New York, NY. Available online at `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 *Proceedings of the European Conference on Object-Oriented Programming* (Cannes, France). Also available as IBM T. J. Watson Research Center Tech. rep. RC 21644.

HANNAN, J. 1995. A type-based analysis for stack allocation in functional languages. In *Proceedings of the 2nd International Static Analysis Symposium*.

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

JavaMemoryModel 2002. Java memory model mailing list. Archived 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 *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*. *SIGPLAN Not. 27*, 6, 235–248.

LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*. *SIGPLAN Not. 23*, 7, 21–34.

LEE, J., PADUA, D., AND MIDKIFF, S. 1999. Basic compiler algorithms for parallel programs. In *Proceedings of 1999 ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*. ACM Press, New York, NY.

LI, Z. AND ABU-SUFAH, W. 1987. On reducing data synchronization in multiprocessed loops. *IEEE Trans. Comput. C-36*, 1 (Jan.), 105–109.

LIANG, D. AND HARROLD, M. J. 1999. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC/FSE'99). 199–215.

MANSON, J. AND PUGH, W. 2001. Core semantics of multithreaded Java. In *Proceedings of the ACM 2001 ISCOPE/Java Grande Conference*. 29–38. A longer version is available as UMCP CS Tech. rep., available online 4215 at `http://www.cs.umd.edu/ pugh/java/memoryModel/semantics.pdf`.

MARLOWE, T. J. 1989. Data flow analysis and incremental iteration. Ph.D. dissertation, Rutgers University, New Brunswick, NJ.

MIDKIFF, S. P. AND PADUA, D. A. 1987. Compiler algorithms for synchronization. *IEEE Trans. Comput. C-36*, 12 (Dec.), 1485–1495.

OLIVER, M., DECIULESCU, E., AND CLARK, C. 2000. Java positioning paper. Available online at `http://www-1.ibm.com/servers/eserver/zseries/software/java/position.html`.

PARK, Y. AND GOLDBERG, B. 1992. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 117–127.

PUGH, W. 1999. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*. 89–98.

REID, A., MCCORQUODALE, J., BAKER, J., HSIEH, W., AND ZACHARY, J. 1999. The need for predictable garbage collection. In *Proceedings of the 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*. 208–218.

RUGGIERI, C. AND MURTAGH, T. 1988. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*. 285–293.

RUGINA, R. AND RINARD, M. C. 1999. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. 77–90.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst. 20*, 1 (Jan.), 1–50.

TOFTE, M. AND TALPIN, J.-P. 1994. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 188–201.

VIVIEN, 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 the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, CO).

WILSON, R. P. AND LAM, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*. *SIGPLAN Not. 30*, 6, 1–12.

Received August 2001; revised January 2003; accepted April 2003