Escape Analysis for Java

Jong-Deok Choi Manish Gupta Mauricio Serrano Vugranam C. Sreedhar Sam Midkiff
IBM T. J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598
{jdchoi, mgupta, mserrano, vugranam, smidkiff}@us.ibm.com

Abstract

This paper presents a simple and efficient data flow algorithm for escape analysis of objects in Java programs to determine (i) if an object can be allocated on the stack; (ii) if an object is accessed only by a single thread during its lifetime, so that synchronization operations on that object can 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 summarized for each method such that the same summary information may be used effectively in different calling contexts. We present an interprocedural algorithm that uses the above property to efficiently compute the connection graph and identify the non-escaping objects for methods and threads. 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 three out of the ten benchmarks (with a median of 19%), 11% to 92% of all lock operations are eliminated in those ten programs (with a median of 51%), and the overall execution time reduction ranges from 2% to 23% (with a median of 7%) on a 333 MHz PowerPC workstation with 128 MB memory.

To appear in the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA99). Denver, Colorado, November 1, 1999.

1 Introduction

Java continues to gain importance as a language for general-purpose computing and for server applications. Performance is an important issue in these application environments. In Java, each object is allocated on the heap and can be deallocated only by garbage collection. Each object has a lock associated with it, which is used to ensure mutual exclusion when a synchronized method or statement is invoked on the object. Both heap allocation and synchronization on locks incur performance overhead.[1] In this paper, we present escape analysis in the context of Java for determining whether an object (1) may *escape the method* (i.e., is not local to the method) that created the object, and (2) may *escape the thread* that created the object (i.e., other threads may access the object).

For Java programs, we identify two important applications of escape analysis:

1. If an object does not escape a method, it can be allocated on the method's stack frame. This has two important implications. First, stack allocation is inherently cheaper than heap allocation, which requires (occasionally) synchronizing the allocator with other threads. Stack allocation also reduces garbage collection overhead, since the storage on the stack is automatically reclaimed when the method returns. As well, if an object does not escape a method, it opens up the possibility of strength-reducing the object accesses and eliminating the creation of the object.

2. If an object does not escape a thread, then no other thread

[1] We use synchronization and synchronization operation synonymously.

accesses the object. This has several benefits, especially in a multithreaded multiprocessor environment. First, we can eliminate the synchronization associated with this object. Note that Java memory model still requires that we flush the Java local memory at `monitorenter` and `monitorexit` statements. Second, objects that are local to a thread can be allocated in the memory of the processor where that thread is scheduled. This local allocation helps improve data locality. Third, with further analysis, some operations to flush the local memory can be safely eliminated.

In this paper, we introduce a new framework for escape analysis, based on a simple program abstraction called the *connection graph*. The connection graph abstraction captures the "connectivity" relationship among heap allocated objects and object references. For escape analysis, we simply perform reachability analysis on the connection graph to determine if an object is local to a method or local to a thread. Different variants of our analysis can be used either in a static Java compiler, a dynamic Java compiler, a Java application extractor, or a bytecode optimizer. To evaluate the effectiveness of our method, we have implemented various flavors of escape analysis in the context of a static Java compiler [11], and have analyzed ten medium to large benchmarks.

The main contributions of this paper are:

- We present a new, simple interprocedural framework (with flow-sensitive and flow-insensitive versions) for escape analysis in the context of Java.
- We demonstrate an important application of escape analysis for Java programs – that of eliminating unnecessary lock operations on thread-local objects. To the best of our knowledge, ours is the first application of escape analysis for eliminating synchronization operations. It leads to significant performance benefits even when using a highly optimized implementation of locks, namely, *thin-locks* [2].
- We describe how to handle *exceptions* in Java, without being unduly conservative. These ideas can be applied to other data flow analyses in the presence of exceptions as well.
- We introduce a simple program abstraction called the connection graph, which is well suited for the purpose of escape analysis. It is different from points-to graphs for alias analysis whose major purpose is memory disambiguation. In the connection graph abstraction, we also introduce the notion of *phantom nodes*, which allows us to summarize the effects of a callee procedure independent of the calling context. This succinct summarization helps improve the overall speed of the algorithm.
- We present extensive experimental results from an implementation of escape analysis in a Java compiler. We show that the compiler is able to detect more than 19% of dynamically created objects as stack-allocatable in five of the ten benchmarks that we examined (finding higher than 70% stack-allocatable objects in three programs). We are able to eliminate 11%-92% of lock operations in those ten programs. The overall performance improvement ranges from 2% to 23% on a 333 MHz IBM PowerPC workstation with 128 MB memory.

The rest of this paper is organized as follows. Section 2 presents our connection graph abstraction. Sections 3 and 4 respectively describe the intraprocedural and interprocedural analyses, to build the connection graph and to identify the objects that do not escape their method or thread of creation. Section 4 also describes the difference between the connection graph for escape analysis and the points-to graph for alias analysis. Section 5 elaborates on handling of Java features like exceptions and object finalizers. Section 6 describes the transformation and the run-time support for the optimization, and Section 7 presents experimental results. Section 8 discusses related work, and finally, Section 9 presents conclusions.

2 Framework for Escape Analysis

We begin by presenting our framework for escape analysis. We first define, in Section 2.1, the notion of escapement and introduce a lattice for escapement. Then in Section 2.2, we introduce a connection graph abstraction for our escape analysis.

2.1 Escapement of an Object

We begin by formalizing the notion of *escapement* of an object from a method or a thread.

Definition 2.1 *Let O be an object instance 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 an object instance and T be a thread (instance). O is said to escape T, again denoted as $Escapes(O, T)$, if another thread, $T' \neq T$, may access O.*

Alternatively, we say that an object O is *stack-allocatable* in M if $\neg Escapes(O, M)$, and an 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'. Thus, we ensure the following proposition:

Proposition 2.3 *For any object O, $\neg Escapes(O, M)$ implies $\neg Escapes(O, T)$, where method M is invoked in thread T.*

Intuitively, the proposition states that an 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.

To aid in our analysis, we define an escapement lattice consisting of three elements: *NoEscape* ($\top$), *ArgEscape*, and *GlobalEscape* ($\perp$). 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, 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 threads and methods). Let $A \in EscapeSet = \{NoEscape, ArgEscape, GlobalEscape\}$, then $A \wedge NoEscape = A$, and $A \wedge GlobalEscape = GlobalEscape$.

Upon the completion of our interprocedural analysis, all objects that are marked *NoEscape* are stack-allocatable in the method in which they are created. Furthermore, all objects that are marked *NoEscape* (due to Proposition 2.3 above) or *ArgEscape*, are local to the thread in which they are created, and so we can eliminate the synchronization in accessing these objects without violating Java semantics.

2.2 Connection Graph Abstraction

In Java, objects are created via `new` statements. To simplify the discussion, we shall view each array as a single, monolithic object. In this section, we introduce a compile-time abstraction called the *Connection Graph* that captures the connectivity relationship among objects.

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

- *N_o represents the set of objects. We create at most one object node per statement.*[2]
- *N_r represents the set of reference variables (locals and formals) in the program.*
- *N_f represents the set of non-static field nodes.*
- *N_g represents the set of static field nodes, i.e., all* global variables *in the program.*
- *E_r is the set of points-to edges. If $x \rightarrow y \in E_r$, then $x \in N_r \cup N_f \cup N_g$ and $y \in N_o$.*
- *E_d is the set of deferred edges. If $x \rightarrow y \in E_d$, then $x, y \in N_r \cup N_f \cup N_g$.*
- *E_f is the set of field edges. If $x \rightarrow y \in E_f$, then $x \in N_o$ and $y \in N_f \cup N_g$.*

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.[3] Also, in our figures, a solid-line edge represents a points-to edge, and a dotted-line edge represents a deferred edge. In the text, we use the notation $x \xrightarrow{P} y$ to represent a points-to edge from node x to node y, $x \xrightarrow{D} y$ to represent a deferred edge from x to y, and $x \xrightarrow{F} y$ to represent a field edge from x to y.

[2] We use a 1-limited naming scheme which creates one node for each `new` statement in the program.

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

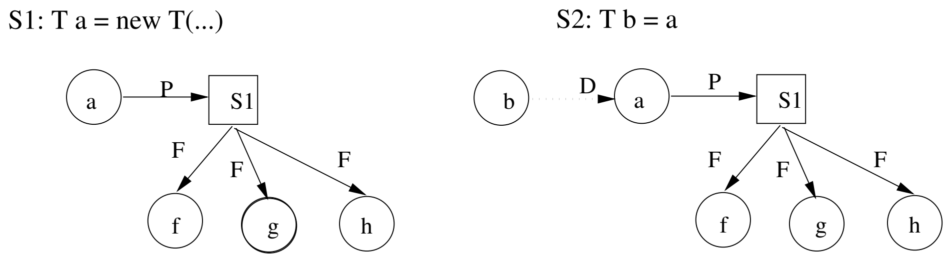


Figure 1: A simple connection graph. Boxes indicate object nodes and circles indicate reference nodes (including field reference nodes). Solid edges indicate points-to edge, dashed edges indicate deferred edges, and edges from boxes to circles indicate field edges.

We assign to each field f in an object a unique number $fid(f)$ that corresponds to the field identifier (or offset) in the class defining the object. Let O_1 and O_2 be two objects constructed from the same class C. Let f be a field defined in C, then $fid(O_1.f) = fid(O_2.f)$.

We use deferred edges to model assignments that merely copy references from one variable to another. Deferred edges defer computations during connection graph construction, and thereby help in reducing the number of graph updates needed during escape analysis. Deferred edges were first introduced for flow-insensitive pointer analysis in [7].

Given a reference node $m \in N_r \cup N_f \cup N_g$, the set of object nodes $O \subseteq N_o$ that it (immediately) points-to can be determined by traversing the deferred edges from m until we visit the first points-to edge in the path. The destination node of the points-to edge will be in O. We formalize this as follows:

Definition 2.5 *Let $m \in N_r \cup N_f \cup N_g$. A points-to path of length one, denoted as $m \stackrel{+P}{\to} n$, is a sequence of edges $m = m_0 \stackrel{D}{\to} m_1 \stackrel{D}{\to} \ldots \stackrel{P}{\to} n$ that terminates in a points-to edge and contains exactly one points-to edge in the path (all other edges, if any, are deferred edges).*

Definition 2.6 *Let $m \in N_r \cup N_f \cup N_g$, then the set of object nodes that nodes m points-to is:*

$$PointsTo(m) = \{n | m \stackrel{+P}{\to} n\}.$$

With each node $n \in N$, we associate an escape state, denoted as $EscapeState[n]$, that is an element of *EscapeSet*. The initial state for each node in N_g is *GlobalEscape*, whereas the initial state for each node in $N_r \cup N_o \cup N_f$, unless otherwise stated, is *NoEscape* (in Sections 4 and 5, we shall discuss nodes representing parameters, thread objects, and objects with non-trivial finalizers, which are initialized as *ArgEscape, GlobalEscape*, and *GlobalEscape*, respectively).

2.3 Basic Idea for Escape Analysis

In the next several sections, we will show how to compute the connection graph abstraction and use it to compute escapement of objects. The intuition behind our algorithm is based on the following key observation: *Let CG be a connection graph for a method M, and let O be an object node in CG. If O can be reached in CG from any node whose escape state is not NoEscape, then O escapes M.* The intuition easily extends to the escapement of an object from a thread.

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.

To simplify our presentation, we introduce a function called $ByPass(p)$ that when applied to a node $p \in N_r \cup N_f$ redi-

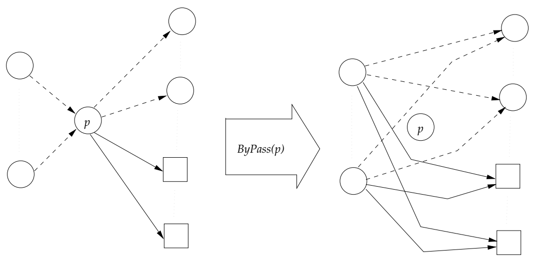


Figure 2: Illustrating $ByPass(p)$ function

rects 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 $\{r \xrightarrow{D} p | r \in R\} \cup \{p \xrightarrow{P} s | s \in S\} \cup \{p \xrightarrow{D} t | t \in T\}$ from the connection graph (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. Note that $ByPass(p)$ can always be applied to a reference node to eliminate its incoming deferred edges.

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

$$\begin{aligned} C_o^s &= f^s(C_i^s) \\ C_i^s &= \wedge_{r \in Pred(s)} C_o^r, \end{aligned}$$

We define a merge between two connection graphs $C_1 = (N_1, E_1)$ and $C_2 = (N_2, E_2)$ to be the union of the two graphs. More formally, $C_1 \wedge C_2 = (N_1 \cup N_2, E_1 \cup E_2)$.

Figure 3 illustrates the connection graphs at various program points computed using the analysis described in this section.[4] Given the bytecode simplification of Java programs, we identify four *basic statements* that affect intraprocedural escape analysis: (i) p = new $\tau()$, (ii) $p = q$, (iii) $p.f = q$, (iv) $p = q.f$. We present the transfer functions for each of these statements.

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.

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

p.f = q Let $U = PointsTo(p)$. If $U = \emptyset$, then either (i) p is null (in which case, a null pointer exception will be thrown), or (ii) the object that p points to was created outside of this method (this could happen if p is a formal parameter or reachable from a formal parameter). We conservatively assume the second possibility (if $U = \emptyset$) and create a *phantom object node* O_{ph}, and insert a points-to edge from p to O_{ph} (if p is null, the edge from p to O_{ph} is spurious, but does not affect the correctness of our analysis).

During interprocedural analysis, the phantom nodes will be mapped back to the actual nodes created by the appropriate procedure. (We also use a 1-limited scheme for creating phantom nodes.) Now let $V = \{v | u \xrightarrow{F} v \textbf{ and } u \in U \textbf{ and } fid(v) = f\}$. Again, it is possible that V is empty. In this case, we create a field reference node (lazily) and add it to V. Finally we add edges in $\{v \xrightarrow{D} q | v \in V\}$ to the connection graph. Note that even for flow-sensitive analysis, we cannot in general kill whatever $p.f$ was pointing to, and so we do not

[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;`.

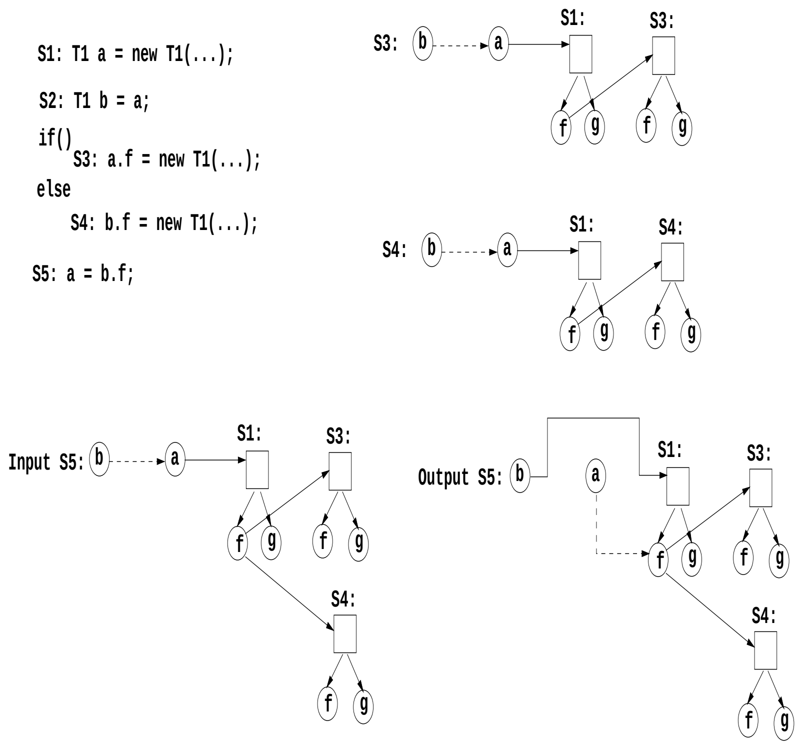


Figure 3: An example illustrating connection graph computation. The connection graphs at S1 and S2 are not shown.

apply $ByPass(p.f)$.[5]

p = q.f Let $U = \{u|q \xrightarrow{+P} u\}$, $V = \{v|u \xrightarrow{F} v \textbf{ and } u \in U \textbf{ and } fid(v) = fid(f)\}$. As in the previous case, if U is empty, we create a phantom node and add it to U, and if V is empty, we create a field reference node and add it to V.

For flow-sensitive analysis, we first apply $ByPass(p)$, and then add the edges in $\{p \xrightarrow{D} v|v \in V\}$ to the connection graph. For flow-insensitive analysis we once again ignore $ByPass(p)$, but add the edges in $\{p \xrightarrow{D} v|v \in V\}$ to the connection graph.

4 Interprocedural Analysis

The intuition behind our interprocedural analysis is based on the following observation. Assume that a method A calls another method B. Now if the method B has already been analyzed for escape analysis, then when A is analyzed intraprocedurally, it can simply use the summary information of B without going through the body of method B (this makes escape analysis different from alias analysis, as described further in Section 4.6). This analysis process is akin to elimination-style of data flow analysis. We use a *program call graph* to represent the caller-callee relation. Since Java supports virtual method calls, we use type information to refine the call graph[6]. We iterate over the nodes in the call graph graph in a reverse topological order until the data flow solution converges.[7]

We handle Java thread objects conservatively. Consider a Java thread object in a method M: $t = new\ Thread();\ t.start()$. $t.start()$ starts the execution of the new thread t. Since the lifetime of t may exceed the lifetime of (an invocation of) M and since the object t is accessed by more than one thread (the creating and the created thread), we mark t as *GlobalEscape*. In general, we mark any object that implements the `Runnable` interface as *GlobalEsape*. This ensures, although conservatively, that any object used as a thread, or any object that is *reachable* from such a thread object globally escapes. Note that this does not mean that objects created *during* the execu-

[5] This is because even a single object that p points to in any k-limited representation may correspond to more than one program object. One can easily construct examples to show that a kill in this case can be incorrect.

[6] We could further refine the call graph by constructing the graph in tandem with the construction of the points-to graph [19].

[7] We ignore back edges in determining the reverse topological order.

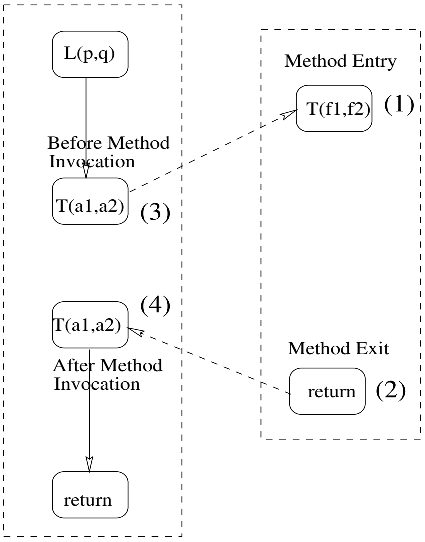
```
class ListElement
{
   int data;
   ListElement next;
   static ListElement g = null;
   ListElement() {data = 0; next = null;}

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

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

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

(A)



(B)

Figure 4: An example program for illustrating interprocedural analysis and its call graph.

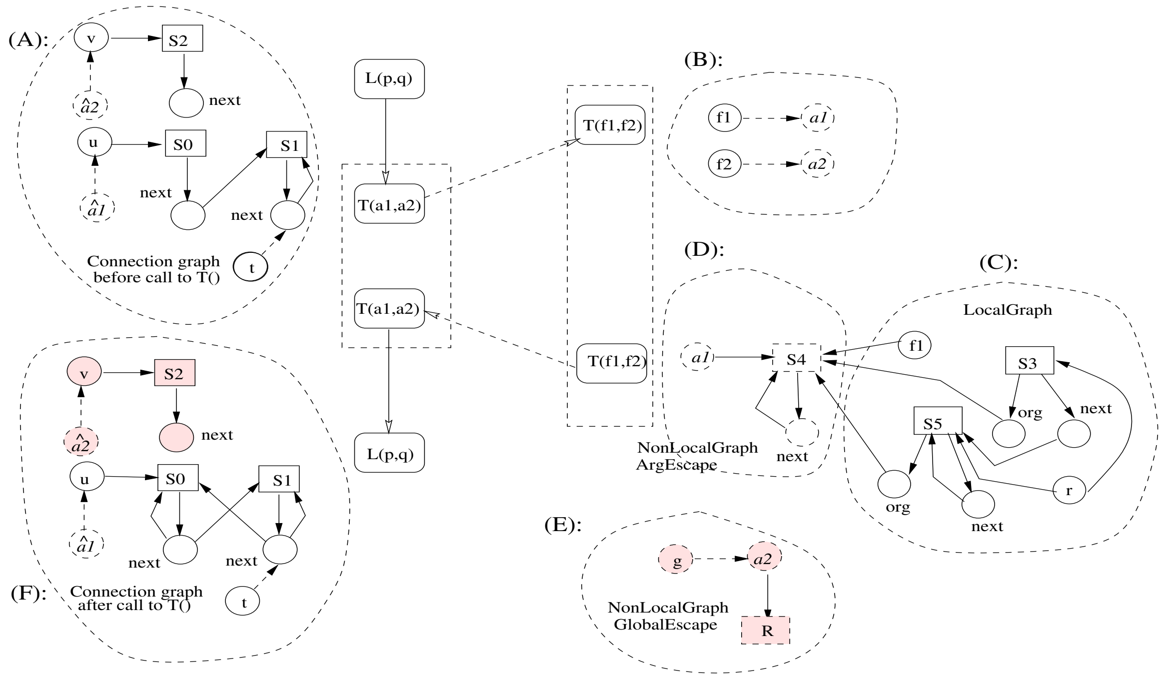


Figure 5: Connection graphs at various points in the call graph. Nodes that escape globally are shadowed.

tion of thread t will be marked *GlobalEsape*.

We will use the Java example shown in Figure 4 to illustrate our interprocedural framework. In this example, method $L()$ constructs a linked list and method $T()$ constructs a tree-like structure. Figure 4(B) shows the caller-callee relation for the example program shown in Figure 4(A). In Figure 4(B) we identify four points of interest to what are relevant for interprocedural analysis: (1) method entry, (2) method exit, (3) immediately before a method invocation, and (4) immediately after a method invocation. We will present our analysis at each of these four points of interest in the following subsections.

4.1 Connection Graph at Method Entry

We process each formal parameter (of reference-type) in a method one at a time. Note that the implicit `this` reference parameter for an instance method appears as the first parameter. For each formal parameter f_i, there exists an actual parameter a_i in the caller of the method that produced the value for f_i. 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 advocates 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 node serves as an anchor for the summary information that will be generated when we finish analyzing the current method.[8] We initialize $EscapeState[f_i] = NoEscape$ and $EscapeState[a_i] = ArgEscape$. Figure 5(B) illustrates the reference nodes `f1` and `f2`, the phantom nodes *a1* and *a2*, and the corresponding deferred edges at the entry of method `T()`.

4.2 Connection Graph at Method Exit

We model a `return` statement that returns a reference to an object as an assignment to a special phantom variable called *return* (similar to formal parameters). Multiple return statements are handled by "merging" their respective return values. After completing intraprocedural escape analysis for a method, we use the *ByPass* function (defined in Section 3) to eliminate all the deferred edges in the CG, creating phantom nodes wherever necessary. For example, the phantom node R in Figure 5(E) is created during this process.

We then do reachability analysis on the CG holding at the return statement of the method to update the escape state of ob-

[8] 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.

jects. 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 4(B).

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 *NonLocalGraph* of the method, and the third subgraph the *LocalGraph*. Figure 6 gives an efficient implementation of the reachability analysis by propagating escape state from nodes with initial state of *GlobalEscape*, then from nodes with initial state of *ArgEscape*. It is easy to show that there can only be edges from *LocalGraph* to *NonLocalGraph*, and not vice versa. The *NonLocalGraph* represents the summary connection graph of the method. This summary information is used at each call site invoking the method, as described below in the next section.[9]

All objects in *LocalGraph* that are created in the current method are marked stack-allocatable. Among the objects (in *NonLocalGraph*) marked *GlobalEscape*, those propagated from a callee of the method need to have their *original nodes* in each callee procedure marked *GlobalEscape*. The original nodes of a propagated node in the current method are identified using the concept of *MapsTo* between two nodes of a caller CG and a callee CG, which is described in Section 4.4. Marking the

[9] As a further optimization to reduce the size of the summary representation, each reference node in *NonLocalGraph* is bypassed by connecting its predecessors directly to its successors, so that the *NonLocalGraph* consists only of the nodes representing actual parameters, objects accessed via the parameters, and a single bottom node.

```
ReachabilityAnalysis()
{
    WorkList = ∅
    /* Nodes in N_g escapes globally */
    foreach node m such that
          EscapeState[m] = GlobalEscape do
       Add m to WorkList.
    while WorkList is not empty do
       Remove a node m from WorkList
       foreach outgoing edge m → n do
          if (EscapeState[n] ≠ GlobalEscape) then
             EscapeState[n] = GlobalEscape
             Add n to WorkList.
          endif
       endfor
    endwhile
    WorkList = ∅
    /* Phantom argument nodes */
    /* state = ArgEscape   */
    foreach node m such that
          EscapeState[m] = ArgEscape do
       Add m to WorkList.
    while WorkList is not empty do
       Remove a node m from WorkList
       foreach outgoing edge m → n do
          if (EscapeState[n] > ArgEscape) then
             EscapeState[n] = ArgEscape
             Add n to WorkList.
          endif
       endfor
    endwhile
}
```

Figure 6: Reachability analysis over connection graph to compute escape state of objects.

original nodes *GlobalEscape* can be performed after the completion of the interprocedural escape analysis in a top down pass over the call graph.

Figure 5(C) - Figure 5(E) 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 5(C) is local to method `T()`, and so will not escape `T()`. We also see that the assignment to the global reference variable, "`g = f2`", makes the formal parameter `f2` and the phantom actual parameter *a2* all *GlobalEscape* as shown in Figure 5(E). (In the figure, a deferred edge from `g` to *a2* is shown for exposition.) The summary graph for method `T()` will consist of the *NonLocalGraph* shown in Figure 5(D). This summary graph will be mapped back to caller's connection graph (see Section 4.4).

4.3 Connection Graph Immediately Before a Method Invocation

At a method invocation site, each parameter passing is handled as an assignment to an actual parameter $\hat{a_i}$ at the caller. Let u_1 be a reference to an object U_1. Consider a call $u_1.foo(u_2, \ldots u_n)$, where $u_2 \ldots u_n$ are actual parameters to $foo()$. We model the call as follows: $\hat{a_1} = u_1; \hat{a_2} = u_2; \ldots; foo(\hat{a_1}, \hat{a_2}, \ldots \hat{a_n})$. Note that if *foo* is a virtual method, we will merge the solution after processing each method to which $u1.foo$ could possibly resolve. Each $\hat{a_i}$ at the call site will be matched with the phantom reference node a_i of the callee method. In Figure 5(A), two nodes, $\hat{a1}$ and $\hat{a2}$, are created with deferred edges pointing to the first and the second actual parameters to the call, `u` and `v`, respectively.

4.4 Connection Graph Immediately After a Method Invocation

At this point, we essentially map the callee's connection graph summary information back to the caller connection graph. Three types of nodes play important role in updating the caller's connection graph (CG) with the callee's CG right after a method invocation: $\hat{a_i}$'s of the caller's CG, a_i's of the callee's CG, and the `return` node of the callee's CG. Updating the caller's CG is done in two steps: (1) updating the node set of the caller's CG using $\hat{a_i}$'s and a_i's; and (2) updating the edge set of the caller's CG using $\hat{a_i}$'s and a_i's. Updating the `return` node is done during the first step by treating the `return` node the same as a_i and treating the target node of the method invocation the same as $\hat{a_i}$.

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 callee CG to caller CG is based on identifying the *MapsTo relation* among object nodes in the two CGs. As a base case, we ensure that a_i maps to $\hat{a_i}$. Given the base case, we also ensure that a node in $PointTo(a_i)$ maps to any node in $PointTo(\hat{a_i})$. We formally define the relation *MapsTo* ($\longmapsto$), among objects belonging to a callee CG and a caller CG recursively as follows:

- $a_i \longmapsto \hat{a_i}$

`UpdateCallerNodes()`
{
27: **foreach** $a_i, \hat{a_i}$ `actual parameter pair` **do**
28: `UpdateNodes` $(a_i, \{\hat{a_i}\})$;
29: **endfor**
}
`UpdateNodes(` f_{ee}`: field node;`
$MapsToF$`: set of field nodes)`
`//` $MapsToF$ `is the set of MapsTo`
`// field nodes of` f_{ee}
{
30: **foreach** object node $n_o \in PointTo(f_{ee})$ **do**
31: **foreach** $\hat{n_o} \in PointTo(f_{er})$
32: `such that` $f_{er} \in MapsToF$ **do**
33: **if** $\hat{n_o} \notin MapsToObj(n_o)$ **then**
34: $MapsToObj(n_o)$
35: $= MapsToObj(n_o) \cup \{\hat{n_o}\}$;
36: **foreach** f'_{ee} `such that` $n_o \xrightarrow{F} f'_{ee}$ **do**
37: $tmpMapsToF = \{f'_{er} \mid \hat{n_o} \xrightarrow{F} f'_{er},$
38: $fid(f'_{ee}) = fid(f'_{er})\}$;
39: `UpdateNodes(` $f'_{ee}, tmpMapsToF$`);`
40: **endfor**
41: **endif**
42: **endfor**
43: **endfor**
}

Figure 7: Algorithm to Update the Caller's Connection Graph Nodes.

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

In Figure 7, $MapsToObj(n)$ denotes the set of objects that n can be mapped to using the above *MapsTo* relation. In the figure, we use the subscript er to denote caller nodes and ee to denote callee nodes. The algorithm starts with a_i and $\hat{a_i}$ as the original "fields" that map to/from each other, and then recursively finds other objects in the caller CG that are *MapsTo* nodes of each corresponding callee object. If there is no *MapsTo* node in the caller CG, we create one with an escape state of *NoEscape*. Then, the escape state of the nodes in $MapsToObj(n)$ is marked *GlobalEscape* if the escape state of n is *GlobalEscape*.

The main body of procedure `UpdateNodes` is applied to all the callee object nodes pointed to by the callee field node f_{ee} (Statement 30). Given a callee object node n_o, Statement 32 computes the set of n_o's *MapsTo* object nodes in the caller graph. This is done by identifying the set of caller object nodes "pointed" to by the caller field node f_{er}, which is itself a *MapsTo* field node of callee node f_{ee} (i.e. $f_{er} \in MapsToF$). A caller object node, $\hat{n_o}$, and its field nodes are created at Statement 32 if no *MapsTo* caller object node exists. Statement 33 is for termination: it skips the body of the loop for $\hat{n_o}$ that is already in $MapsToObj(n_o)$. Given a callee object node n_o and its *MapsTo* caller node $\hat{n_o}$, Statement 38 computes, for each field node of n_o (i.e. $f_{ee}^{'}$), the set of *MapsTo* field nodes of the caller (i.e. $tmpMapsToF$). It then recursively invokes `UpdateNodes`, passing $f_{ee}^{'}$ and $tmpMapsToF$ as the new parameters (Statement 39).

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 33 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 $\hat{p} \xrightarrow{F} \hat{f_p} \xrightarrow{P} \hat{q}$ by inserting a points-to edge $\hat{f_p} \xrightarrow{P} \hat{q}$ for each field node $\hat{f_p}$ of $\hat{p}$ such that $fid(f_p) = fid(\hat{f_p})$.

Example

Consider the summary connection graph *NonLocalGraphs* shown in Figure 5(D) and Figure 5(E). First, all nodes that are reachable from global variable `g` are marked $\perp$ (or *GlobalEscape*). Then, all nodes reachable from the phantom node *a1*, but not reachable from `g` are marked as *ArgEscape*. Now when we analyze method `L()` intraprocedurally we would construct the connection shown in figure that is right after the method site of `T()`. We will first mark the phantom node *a1* of the callee (in Figure 5(D)) and the phantom node $\hat{a1}$ of the caller (in Figure 5(F)) as the initial "field" nodes. Then we will map the phantom node `S4`, pointed to by *a1*, to `S0`, pointed to by $\hat{a1}$. The cycle in the *NonLocalGraph* of `T()` results in also mapping `S1` as a *MapsTo* node of `S4`. The cycle also results in inserting edges from the `next` fields of `S0` and `S1` to both `S0` and `S1`. This is a result of the 1-limited approach we take in creating a phantom node: we create at most one phantom node at a statement for each type. Now since *a2* is marked $\perp$, all the nodes of the caller reachable from $\hat{a2}$ will also be marked as $\perp$.

4.5 Java's Strong Type Information

We can exploit Java's strong type system in computing the connection graph for a method whose body cannot be (or, has not been) analyzed. The representation for such a method, called a *bottom method*, is called the *bottom graph*, which has one node for each class of the program that has been instantiated. Given two nodes N_1 and N_2 in the bottom graph that represent two classes C_1 and C_2, respectively, there is a points-to edge from N_1 to N_2 if C_1 contains a field that is a reference to C_2. There is a deferred edge from N_1 to N_2 if C_2 is a sub-type of C_1. In effect, the bottom graph is the most conservative connection graph of the program allowed under Java's type system. The bottom graph can be used to (conservatively) establish connections among nodes that are reachable from the actual parameters passed to a bottom method. Examples of bottom methods are native methods implemented in a non-Java language. Our current implementation does not take advantage of the type information in bottom methods.

In a dynamic optimization system, a method that has not been analyzed and optimized by the compiler also becomes a *bottom method* when the compiler generates code for a caller of the method. In this case, the bottom method may have been interpreted or compiled without analysis/optimization. The combination of the bottom graph and the summary graph makes our approach for escape analysis well suited for dynamic Java compilation systems such as Jalapeño at IBM Research [8].

4.6 Escape Analysis versus Points-to Analysis

Connection graph for escape analysis and *points-to graphs* for pointer-induced alias analysis [16, 19] are similar to each other in that both are static abstractions of dynamic data structures with pointers (or references in Java). The main goal of alias analysis, however, is *memory disambiguation* to answer the question whether two reference (pointer) expressions (of the form `a.b.c.d...`) can resolve to the same memory location during execution. The points-to graph, for correctness, should lead to the same node in the graph if the two reference expressions might resolve to the same memory location during execution.

The main goal of escape analysis, on the other hand, is to identify objects that might escape a (dynamic) scope such as a method invocation or a thread object. The connection graph may lead to different nodes in the graph for two pointer expressions that might resolve to the same memory location, and can still be correct. We can, therefore, safely ignore the calling context for escape analysis, although not for pointer analysis.

5 Handling Exceptions and Finalization of Java

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

5.1 Exceptions

We now show how our framework handles exceptions. Exceptions are *precise* in Java, hence any code motion across the exception point should be invisible to the user program. An exception thrown by a statement is caught by the closest dynamically enclosing `catch` block that handles the exception [17].

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. The *factored control flow graph (FCFG)* of the Jalapeño dynamic optimizing compiler for Java does not add these edges physically in the control flow graphs, but still allows for correctly identifying the potential control flows due to exceptions [9].

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

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

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

A method (transitively) invoked within a `try-catch` block can be handled in the same manner as a regular statement block in its place: we can kill any locals declared within the nested block, be it a regular statement block or a method block. 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 [17]. The class `Object`, which is a superclass of every other class, provides a default definition of the `finalize` method, which takes no action. If a class overrides the `finalize` method such that its `this` parameter is referenced, it means that an object of that class is reachable (due to the invocation of the finalizer) even after there are no more references to it from any live thread. We deal with this problem by marking each object of the class overriding the finalizer as *GlobalEscape* ($\perp$).

6 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 [11]: (1) allocation of objects on the stack, and (2) elimination of unnecessary synchronization operations. In this section, we describe the transformations applied to the user code (based on the analysis described in previous sections) and the run-time support to implement these optimizations.

6.1 Transformation

Once the analysis converges during the iteration over the call graph (i.e., when there are no further changes being made to any connection graph in terms of edges or the *EscapeState* of nodes), we mark each `new` site in the program as follows, based on the following information: (i) if the *EscapeState* of the corresponding object node is *NoEscape*, the `new` site is marked stack-allocatable, and (ii) 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` essentially grows the current stack frame at run time by some amount. In our current implementation, we do not reuse the space allocated by `alloca`, even if that space is no longer live. [10]

Program	Description
vtrans	High Performance Java Translator (IBM)
jgl	Java Generic Library 1.0 (ObjectSpace)
jacorb	Java Object Request Broker 0.5 (U. Freie)
jolt	Java to C translator (KB Sriram)
jobe	Java Obfuscator 1.0 (E. Jokipii)
javacup	Java Constructor of Parsers (S. Hudson)
hashjava	Java Obfuscator (KB Sriram)
toba	Java to C translator (U. Arizona)
wingdis	Java decompiler, demo version (WingSoft)
pbob	portable Business Object Benchmark (IBM)

Table 1: Benchmarks used in our experiments.

A secondary benefit of stack allocation is the elimination of occasional synchronization for allocation of objects from the thread-common heap. In order to avoid synchronization on each heap allocation, the run-time system in HPCJ uses the following scheme. Each thread usually allocates objects from its thread-local heap space. For allocating a large object or when the local heap space is exhausted, the thread needs to allocate from thread-common heap space, which requires a relatively heavy-weight synchronization. Stack-allocated objects reduce the requirement for allocations from the thread-common heap space.

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

We mark objects at the allocation sites using a single bit in the object representation, indicating whether the object is thread-local. At the use sites of objects, we modified the routine implementing *monitorenter* on an object to bypass the expensive atomic operation (`compare_and_swap`) if its thread-

[10] In cases where (i) the object requires a fixed size, and (ii) 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 take advantage of this special case.

local bit is set, and instead use a non-atomic operation. It is important to note that our scheme has benefits even for the thin-lock synchronization implementation [2], which still needs an atomic operation (`compare_and_swap`); we completely eliminate the need for atomic lock operations for thread-local objects. Note that we still flush the local memory to ensure that global variables are made visible at synchronization points to observe Java semantics [17]. 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.

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. 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 $(k \bmod m)$th field node. In our implementation, we used $m = 3$. Bounding the number of fields reduces the space and time requirement for our analysis, but can make the result less precise.

Our testbed consisted of a 333 MHz uniprocessor PowerPC running AIX 4.1.5, with 1 MB L2 Cache and 512 MB memory. We selected a set of 10 medium-sized to large-sized benchmarks described in Table 1 to run our experiments. Table 2 gives the relevant characteristics for the benchmark programs. Columns 2 and 3 give the number of classes and the size of the classes in bytes for the set of programs. Columns 4 and 5 present the total number of objects dynamically allocated in the user code and overall (including both the user code and the library code). Columns 6 and 7 show the cumulative space in bytes occupied by the objects during program execution. Finally, columns 8 and 9 show the total number of lock operations dynamically encountered during execution.

In the rest of this section, we present our results for the above variants of our analysis. All of the remaining measurements that we present refer to objects created in the user code alone. Modifying any operations related to object creation in the library code would require recompilation of the library code (not done in our current implementation). 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). We did not see much difference between FS and FI (i.e. flow-sensitive and flow-insensitive without bounding the number of fields distinguished). And in most cases, bounding the number of field did not make much difference in the percentage values (for example, see `trans`, `jgl`, `jolt`, `jobe`, `javacup`, `hashjava`, and `wingdis`). Interestingly, `toba` and `jolt` (both of which are Java to C translators) have similar characteristics in terms of stack allocatability of objects. Both of these benchmarks have a substantial number of objects that are stack-allocatable. But in the case of `toba`, limiting the number of fields drastically reduces the number of objects that are stack-allocatable.

7.2 Lock Elimination

For lock elimination, we collected two sets of data (again for different variants of the analysis). First we measured the number of dynamic objects that are thread-local and then we measured how many lock operations are executed over these objects. Figure 10 shows the percentage of user objects that are local to a thread, and Figure 11 shows the percentage of lock operations that are removed for these thread-local objects during execution. It can be seen that our most precise analysis ver-

Program	Number of classes	size of classes	Number of objects allocated		Size of objects in bytes allocated		Total number of locks	
			user	user + library	user	user + library	user	user + library
trans	142	503K	263K	727K	7656K	31333K	868K	885K
jgl	135	217K	3808K	4157K	124409K	139027K	10391K	10434K
jacorb	436	308K	103K	48036K	2815K	3423323K	546K	672K
jolt	46	90K	94K	593K	3006K	17511K	1030K	1348K
jobe	46	60K	204K	339K	7957K	13331K	77K	106K
javacup	59	101K	67K	330K	1672K	8454K	191K	287K
hashjava	98	183K	173K	248K	4671K	827K	158K	165K
toba	19	86K	154K	2201K	5878K	59356K	1060K	1246K
wingdis	48	178K	840K	2561K	25902K	92238K	2105K	2299K
pbob	65	333K	19787K	48206K	639980K	2749520K	35691K	171189K

Table 2: Benchmarks characteristics

sion finds a lot of opportunities to eliminate synchronization, removing more than 50% of the synchronization operations in half of the programs. One can deduce certain interesting characteristics by comparing the two graphs. For `pbob`, one can see that the percentage of thread-local objects ($\approx 50\%$) is higher than the percentage of locks removed ($\approx 15\%$). Our observation is that relatively few thread-local objects are actually involved in synchronization.

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

7.3 Execution Time Improvements

Table 3 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. `pbob` was run on a 4-way PowerPC SMP machine.

Table 3 shows an appreciable performance improvement (greater than 15% reduction in execution time) in three programs and relatively modest improvements in other programs.

8 Related Work

Lifetime analysis of dynamically allocated objects has been traditionally used for compile time storage management [24, 22, 3]. Park and Goldberg introduced the term *escape analysis* [22] for statically determining which parts of a list 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 [12, 4]. Birkedal et al. [3] propose a region allocation model, where regions are managed during compilation. A type system is used to translate a functional program to another functional program annotated with regions where values could be stored. Hanan [18] uses a type system to translate a strongly typed functional program to an annotated functional program, where the annotation is used for for stack allocation rather than for region allocation.

Prior work on synchronization optimization has addressed the problem of reducing the amount of synchronization [13, 20, 21]. These approaches assume that the mutual exclusion

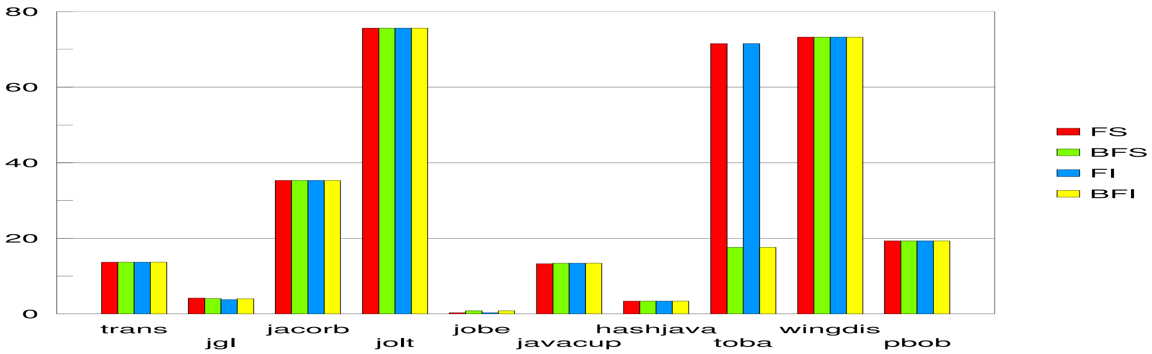


Figure 8: Percentage of user local objects allocated on the stack.

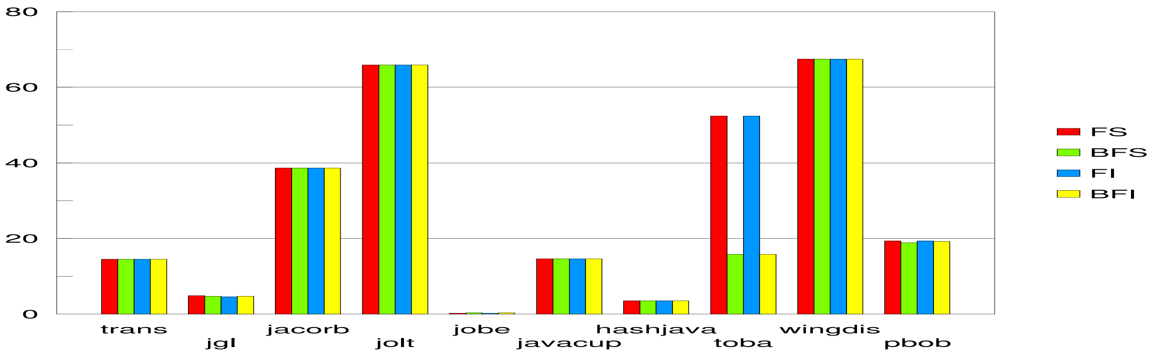


Figure 9: Percentage of user local object space allocated on the stack.

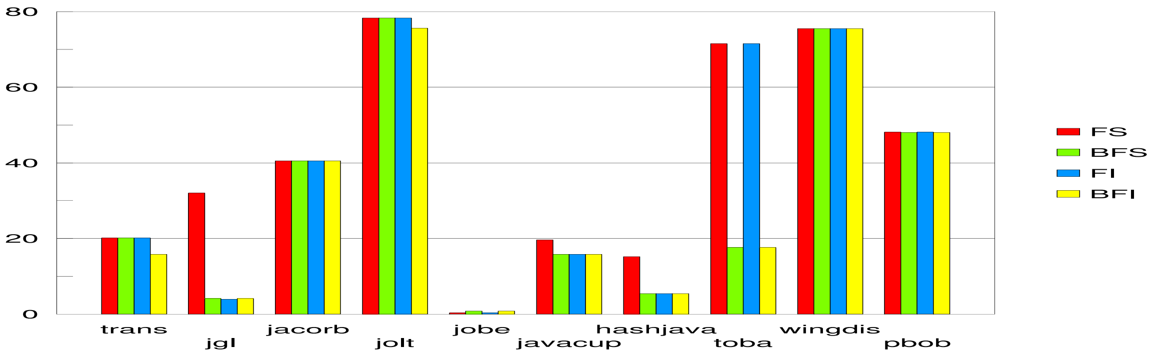


Figure 10: Percentage of thread local objects.

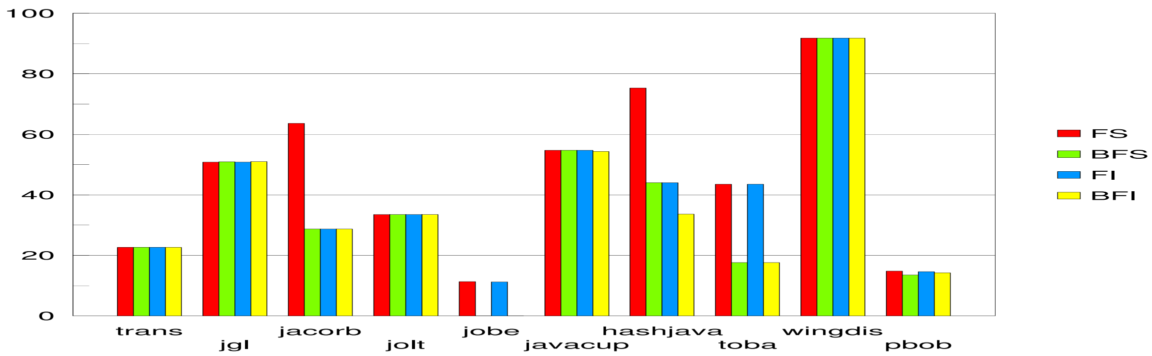


Figure 11: Percentage of locks removed.

ordering implied by the original synchronization is needed, and so only attempt to reduce the number of such operations without violating the original ordering. In contrast, our approach finds unnecessary mutual exclusion lock operations and eliminates them.

There have been a number of parallel efforts on escape analysis for Java [15, 23, 6, 1, 25, 5]. Bogda and Hölzle use set constraints for computing thread-local objects [6]. Their system is a bytecode translater, and uses replication of execution paths as the means for eliminating unnecessary synchronization. After replication, they convert synchronized methods that access only thread-local objects into non-synchronized methods. This conversion, in general, breaks Java semantics—since at the beginning and the end of a synchronized method or a statement, the local memory has to be synchronized with the main memory (see Section 6). Replication, however, offers an opportunity for specializing an allocation site that generates both thread-local and thread-global objects along different call chains. They also summarize the effect of native methods (although manually). Using the summary information, they improve the precision of their analysis. Our approach can be extended to include specialization and native method analysis.

Aldrich et al. describe a set of analyses for eliminating unnecessary synchronization on multiple re-entries of a monitor by the same thread, nested monitors, and thread-local objects [1]. They also remove synchronization operations, which can break Java semantics. They claim that their approach, however, should be safe for most well-written multithreaded programs in Java, which assume a "looser synchronization" model than what Java provides.

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

Table 3: Improvements in execution time

Blanchet uses type heights (which are integer values) to encode how an object of one type can have references to other objects or is a subtype of another object [5]. The escaping part of an object is represented by the height of its type. He proposes a two-phase (a backward phase and a forward phase) flow-insensitive analysis for computing escape information. He uses escape analysis, like our work, for both stack allocation and synchronization elimination. For synchronization elimination, before acquiring a lock on an object o, his algorithm tests at runtime whether o is on the stack – if it is, the synchronization is skipped. Our algorithm uses a separate thread-local bit within each object, and can skip the synchronization even for objects that are not stack allocatable (but are thread local).

To reduce the size of finite-state models of concurrent Java programs, Corbett uses a technique called virtual coarsening [10]. In virtual coarsening, invisible actions (e.g., updates to variables that are local or protected by a lock) are collapsed into adjacent visible actions. Corbett uses a simple intraprocedural pointer analysis (after method inlining) to identify the heap objects that are local to a thread, and also to identify the variables that are guarded by various locks. Dolby's analysis technique for inlining of objects in C++ can also be extended to eliminate synchronization in Java programs [14].

9 Conclusions

In this paper, we have presented a new interprocedural algorithm for escape analysis. Apart from using escape analysis for stack allocation of objects, we have demonstrated an important new application of escape analysis – eliminating unnecessary synchronization in Java programs. Our approach uses a data flow analysis framework and maps escape analysis to a simple reachability problem over a connection graph abstraction. With a preliminary implementation of this algorithm, our static Java compiler is able to detect a significant percentage of dynamically created objects as stack-allocatable, as high as 70% in some cases. It is able to eliminate 11% to 92% of lock operations in our benchmarks (eliminating more than 50% of lock operations in half of them). We observe overall performance improvements ranging from 2% to 23% on our benchmarks, and find that most of these improvements come from savings on lock operations on the thread-local objects, as these programs do not seem to incur a significant garbage collection overhead due to relatively low memory usage. We expect to improve these results with a more aggressive implementation of our algorithm that treats native methods less conservatively, and by applying our optimizations to the Java standard class library routines as well. In the future, we also plan to extend our algorithm to cover the more general problem of region-based storage allocation, and to eliminate unnecessary `sync` operations for flushing of local memory.

Interprocedural analysis in the presence of dynamic loading and reloading of classes, as allowed in Java, is in general a hard problem. We are currently working on extending our escape analysis to Jalapeño, a dynamic Java compilation system at IBM Research [8].

Acknowledgement

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

References

[1] Jonathan Alridch, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analysis for eliminating unnessary synchronization from java programs. In *Proceedings of the Sixth International Static Analysis Symposium*, Venezia, Italy, September 1999.

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

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

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

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

[6] Jeff Bodga and Urs Hölzle. Removing unnecessay synchronization in java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.

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

[8] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for java. In *Proc. ACM SIGPLAN 1999 Java Grande Conference*, June 1999.

[9] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs, 1999. To appear at PASTE '99.

[10] James C. Corbett. Constructing compact models of concurrent java programs. In *Proceedings of the 1998 International Symposium of Software Testing and Analysis*. ACM Press, March 1998.

[11] IBM Corporation. IBM High Performance Compiler for Java, 1997. Information available in Web page at `http://simont01.torolab.ibm.com/hpj/hpj.html`, available for download at `http://www.alphaWorks.ibm.com/formula`.

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

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

[14] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN Conference of Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[15] David Gay and Bjarne Steensgaard. Stack allocating objects in Java. Research Report, Microsoft Research, 1999.

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

[17] James Gosling, Bill Joy, and Guy Steele. *The Java*$^{(\mathrm{TM})}$ *Language Specification*. Addison-Wesley, 1996.

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

[19] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*. To appear.

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

[21] S.P. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[22] Y.G. Park and B. Goldberg. Escape analysis on lists. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–127, July 1992.

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

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

[25] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.