Incrementalized Pointer and Escape Analysis*

Frédéric Vivien
ICPS/LSIIT
Université Louis Pasteur
Strasbourg, France
vivien@icps.u-strasbg.fr

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@lcs.mit.edu

ABSTRACT

We present a new pointer and escape analysis. Instead of analyzing the whole program, the algorithm incrementally analyzes only those parts of the program that may deliver useful results. An analysis policy monitors the analysis results to direct the incremental investment of analysis resources to those parts of the program that offer the highest expected optimization return.

Our experimental results show that almost all of the objects are allocated at a small number of allocation sites and that an incremental analysis of a small region of the program surrounding each site can deliver almost all of the benefit of a whole-program analysis. Our analysis policy is usually able to deliver this benefit at a fraction of the whole-program analysis cost.

1. INTRODUCTION

Program analysis research has focused on two kinds of analyses: *local* analyses, which analyze a single procedure, and *whole-program* analyses, which analyze the entire program. Local analyses fail to exploit information available across procedure boundaries; whole-program analyses are potentially quite expensive for large programs and are problematic when parts of the program are not available in analyzable form.

This paper describes our experience *incrementalizing* an existing whole-program analysis so that it can analyze arbitrary regions of complete or incomplete programs. The new analysis can 1) analyze each method independently of its caller methods, 2) skip the analysis of potentially invoked methods, and 3) incrementally incorporate analysis results from previously skipped methods into an existing analysis result. These features promote a structure in which the algorithm executes under the direction of an analysis policy. The policy continuously monitors the analysis results to direct the incremental investment of analysis resources to those parts of the program that offer the most attractive return (in terms of optimization opportunities) on the invested resources. Our experimental results indicate that this approach usually delivers almost all of the benefit of the whole-program analysis, but at a fraction of the cost.

1.1 Analysis Overview

Our analysis incrementalizes an existing whole-program analysis for extracting points-to and escape information [16]. The basic abstraction in this analysis is a points-to escape graph. The nodes of the graph represent objects; the edges represent references between objects. In addition to points-to information, the analysis records how objects escape the currently analyzed region of the program to be accessed by unanalyzed regions. An object may escape to an unanalyzed caller via a parameter passed into the analyzed region or via the return value. It may also escape to a potentially invoked but unanalyzed method via a parameter passed into that method. Finally, it may escape via a global variable or parallel thread. If an object does not escape, it is captured.

The analysis is flow sensitive, context sensitive, and compositional. Guided by the analysis policy, it performs an incremental analysis of the neighborhood of the program surrounding selected object allocation sites. When it first analyzes a method, it skips the analysis of all potentially invoked methods, but maintains enough information to reconstruct the result of analyzing these methods should it become desirable to do so. The analysis policy then examines the graph to find objects that escape, directing the incremental integration of (possibly cached) analysis results from potential callers (if the object escapes to the caller) or potentially invoked methods (if the object escapes into these methods). Because the analysis has complete information about captured objects, the goal is to analyze just enough of the program to capture objects of interest.

1.2 Analysis Policy

We formulate the analysis policy as a solution to an investment problem. At each step of the analysis, the policy can invest analysis resources in any one of several allocation sites in an attempt to capture the objects allocated at that site. To invest its resources wisely, the policy uses empirical data from previous analyses, the current analysis result for each site, and profiling data from a previous training run to estimate the marginal return on invested analysis resources for each site.

During the analysis, the allocation sites compete for resources. At each step, the policy invests its next unit of analysis resources in the allocation site that offers the best

*The full version of this paper is available at www.cag.lcs.mit.edu/~rinard/paper/pldi01.full.ps. This research was done while Frédéric Vivien was a Visiting Professor in the MIT Laboratory for Computer Science. The research was supported in part by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-63513.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2001 ACM 0-89791-88-6/97/05 ...$5.00.

marginal return. When the unit expires, the policy recomputes the estimated returns and again invests in the (potentially different) allocation site with the best estimated marginal return. As the analysis proceeds and the policy obtains more information about each allocation site, the marginal return estimates become more accurate and the quality of the investment decisions improves.

1.3 Analysis Uses

We use the analysis results to enable a stack allocation optimization. If the analysis captures an object in its allocating method, the object is unreachable once the method returns. In this case, the generated code allocates the object in the activation record of its allocating method. If the object escapes the allocating method, but is captured in one or more of the methods that directly invoke the allocating method, the compiler inlines the allocating method into the capturing callers, then generates code to allocate the captured objects in the activation record of the caller. The success of this optimization depends on the characteristics of the application. The vast majority of the objects in our benchmark applications are allocated at a small subset of the allocation sites. For some applications the analysis is able to capture and stack allocate all of the objects allocated at these sites. In other applications these objects escape and the analysis finds few relevant optimization opportunities.

Other optimization uses include synchronization elimination, the elimination of `ScopedMemory` checks in Real-Time Java [6], and a range of traditional compiler optimizations. Potential software engineering uses include the evaluation of programmer hypotheses regarding points-to and escape information for specific objects, the discovery of methods with no externally visible side effects, and the extraction of information about how methods access data from the enclosing environment.

Because the analysis is designed to be driven by an analysis policy to explore only those regions of the program that are relevant to a specific analysis goal, we expect the analysis to be particularly useful in settings (such as dynamic compilers and interactive software engineering tools) in which it must quickly answer queries about specific objects.

1.4 Context

In general, a base analysis must have several key properties to be a good candidate for incrementalization: it must be able to analyze methods independently of their callers, it must be able to skip the analysis of invoked methods, and it must be able to recognize when a partial analysis of the program has given it enough information to apply the desired optimization. Algorithms that incorporate escape information are good candidates for incrementalization because they enable the analysis to recognize captured objects (for which it has complete information). As discussed further in Section 7, many existing escape analyses either have or can easily be extended to have the other two key properties [14, 7, 3]. Many of these algorithms are significantly more efficient than our base algorithm, and we would expect incrementalization to provide these algorithms with additional efficiency increases comparable to those we observed for our algorithm. Compiler developers would therefore be able to choose from a variety of efficient analyses, with some analyses imposing little to no overhead.

An arguably more important benefit is the fact that incrementalized algorithms usually analyze only a local neighborhood of the program surrounding each object allocation site. The analysis time for each site is therefore independent of the overall size of the program, enabling the analysis to scale to handle programs of arbitrary size. And incrementalized algorithms can analyze incomplete programs.

1.5 Contributions

This paper makes the following contributions:

- **Analysis Approach:** It presents an incremental approach to program analysis. Instead of analyzing the entire program, the analysis is focused by an analysis policy to incrementally analyze only those regions of the program that may provide useful results.
- **Analysis Algorithm:** It presents a new combined pointer and escape analysis algorithm based on the incremental approach described above.
- **Analysis Policy:** It formulates the analysis policy as a solution to an investment problem. Presented with several analysis opportunities, the analysis policy incrementally invests analysis resources in those opportunities that offer the best estimated marginal return.
- **Experimental Results:** Our experimental results show that, for our benchmark programs, our analysis policy delivers almost all of the benefit of the whole-program analysis at a fraction of the cost.

The remainder of the paper is structured as follows. Section 2 presents several examples. Section 3 presents our previously published base whole-program analysis [16]; readers familiar with this analysis can skip this section. Section 4 presents the incrementalized analysis. Section 5 presents the analysis policy; Section 6 presents experimental results. Section 7 discusses related work; we conclude in Section 8.

2. EXAMPLES

We next present several examples that illustrate the basic approach of our analysis. Figure 1 presents two classes: the `complex` class, which implements a complex number package, and the `client` class, which uses the package. The `complex` class uses two mechanisms for returning values to callers: the `add` and `multiplyAdd` methods write the result into the receiver object (the `this` object), while the `multiply` method allocates a new object to hold the result.

2.1 The `compute` Method

We assume that the analysis policy first targets the object allocation site at line `3` of the `compute` method. The goal is to capture the objects allocated at this site and allocate them on the call stack. The initial analysis of `compute` skips the call to the `multiplyAdd` method. Because the analysis is flow sensitive, it produces a points-to escape graph for each program point in the `compute` method. Because the stack allocation optimization ties object lifetimes to method lifetimes, the legality of this optimization is determined by the points-to escape graph at the end of the method.

Figure 2 presents the points-to escape graph from the end of the `compute` method. The solid nodes are *inside* nodes, which represent objects created inside the currently analyzed region of the program. Node `3` is an inside node

```
class complex {
  double x,y;
  complex(double a, double b) { x = a; y = b; }
  void add(complex u, complex v) {
    x = u.x+v.x; y = u.y+v.y;
  }
  complex multiply(complex m) {
11: complex r = new complex(x*m.x-y*m.y, x*m.y+y*m.x);
    return(r);
  }
  void multiplyAdd(complex a, complex b, complex c) {
    complex s = b.multiply(c);
    this.add(a, s);
  }
}
class client {
  public static void compute(complex d, complex e) {
3:  complex t = new complex(0.0, 0.0);
    t.multiplyAdd(d,e,e);
  }
}
```

Figure 1: Complex Number and Client Classes

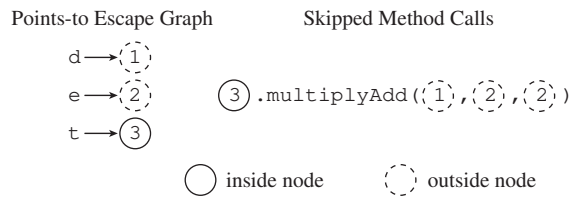

Figure 2: Analysis Result from `compute` Method

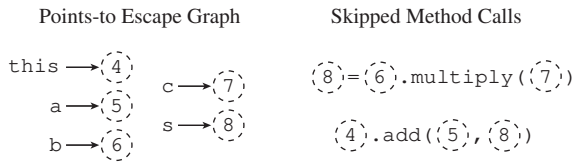

Figure 3: Analysis Result from `multiplyAdd` Method

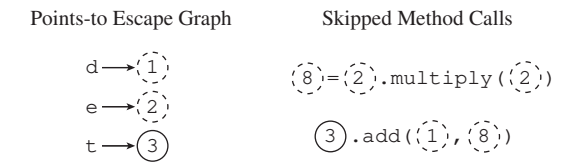

Figure 4: Analysis Result from `compute` Method after Integrating Result from `multiplyAdd`

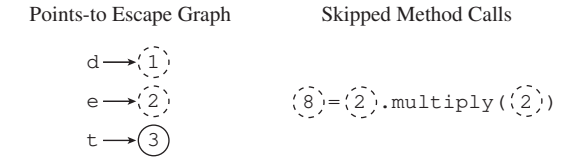

Figure 5: Analysis Result from `compute` Method after Integrating Results from `multiplyAdd` and `add`

that represents all objects created at line 3 in the `compute` method. The dashed nodes are *outside* nodes, which represent objects not identified as created inside the analyzed region. Nodes 1 and 2 are a kind of outside node called a *parameter* node; they represent the parameters to the `compute` method. The analysis result also records the skipped call sites and the actual parameters at each site.

In this case, the analysis policy notices that the target node (node 3) escapes because it is a parameter to the skipped call to `multiplyAdd`. It therefore directs the algorithm to analyze the `multiplyAdd` method and integrate the result into the points-to escape graph from the program point at the end of the `compute` method.

Figure 3 presents the points-to escape graph from the initial analysis of the `multiplyAdd` method. Nodes 4 through 7 are parameter nodes. Node 8 is another kind of outside node: a *return* node that represents the return value of an unanalyzed method, in this case the `multiply` method. To integrate this graph into the caller graph from the `compute` method, the analysis first maps the parameter nodes from the `multiplyAdd` method to the nodes that represent the actual parameters at the call site. In our example, node 4 maps to node 3, node 5 maps to node 1, and nodes 6 and 7 both map to node 2. The analysis uses this mapping to combine the graphs into the new graph in Figure 4. The analysis policy examines the new graph and determines that the target node now escapes via the call to the `add` method. It therefore directs the algorithm to analyze the `add` method and integrate the resulting points-to escape graph into the current graph for the `compute` method. Note that because the call to the `multiply` method has no effect on the escape status of the target node, the analysis policy directs the algorithm to leave this method unanalyzed.

Figure 5 presents the new graph after the integration of the graph from the `add` method. Because the `add` method does not change the points-to or escape information, the net effect is simply to remove the skipped call to the `add` method. Note that the target node (node 3) is captured in this graph, which implies that it is not accessible when the `compute` method returns. The compiler can therefore generate code that allocates all objects from the corresponding allocation site in the activation record of this method.

2.2 The `multiply` Method

The analysis next targets the object allocation site at line 11 of the `multiply` method in Figure 1. Figure 6 presents the points-to escape graph from this method, which indicates that the target node (node 11) escapes to the caller (in this case the `multiplyAdd` method) via the return value. The algorithm avoids repeated method reanalyses by retrieving the cached points-to escape graph for the `multiplyAdd` method, then integrating the graph from the `multiply` method into this cached graph. Figure 7 presents the resulting points-to escape graph, which is cached as the new (more precise) points-to escape graph for the `multiplyAdd` method. This graph indicates that the target node (node 11) does not escape to the caller of the `multiplyAdd` method, but does escape via the unanalyzed call to the `add` method. The analysis therefore retrieves the cached points-to escape graph from the `add` method, then integrates this graph into the current graph from the `multiplyAdd` method. Figure 8 presents the resulting graph. Once again, the algorithm caches this result as the new graph for the `multiplyAdd` method. The target

node (node 11) is captured in this graph — it escapes its enclosing method (the `multiply` method), but is recaptured in a caller (the `multiplyAdd` method).

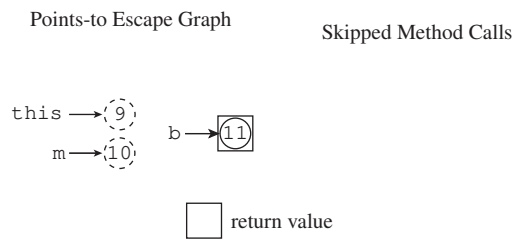

Figure 6: Analysis Result from `multiply` Method

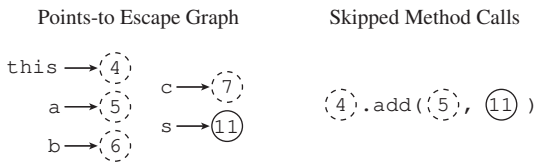

Figure 7: Analysis Result from `multiplyAdd` Method after Integrating Result from `multiply` Method

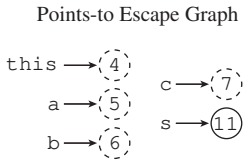

Figure 8: Analysis Result from `multiplyAdd` Method after Integrating Results from `multiply` and `add`

At this point the compiler has several options: it can inline the `multiply` method into the `multiplyAdd` method and allocate the object on the stack, or it can preallocate the object on the stack frame of the `multiplyAdd` method, then pass it in by reference to a specialized version of the `multiply` routine. Both options enable stack allocation even if the node is captured in some but not all invocation paths, if the analysis policy declines to analyze all potential callers, or if it is not possible to identify all potential callers at compile time. Our implemented compiler uses inlining.

2.3 Object Field Accesses

Our next example illustrates how the analysis deals with object field accesses. Figure 9 presents a rational number class that deals with return values in yet another way. Each `Rational` object has a field called `result`; the methods in Figure 9 that operate on these objects store the result of their computation in this field for the caller to access.

We next discuss how the analysis policy guides the analysis for the `Rational` allocation site at line `1` in the `evaluate` method. Figure 10 presents the initial analysis result at the end of this method. The dashed edge between nodes `1` and `2` is an *outside* edge, which represents references not identified as created inside the currently analyzed region of the program. Outside edges always point from an escaped node to a new kind of outside node, a *load* node, which represents objects whose references are loaded at a given load state-

```
class Rational {
  int numerator, denominator;
  Rational result;
  Rational(int n, int d) {
    numerator = n;
    denominator = d;
  }
  void scale(int m) {
    result = new Rational(numerator * m,
                denominator);
  }
  void abs() {
    int n = numerator;
    int d = denominator;
    if (n < 0) n = -n;
    if (d < 0) d = -d;
    if (d % n == 0) {
 4:   result = new Rational(n / d, 1);
    } else {
 5:   result = new Rational(n, d);
    }
  }
}
class client {
  public static void evaluate(int i, int j) {
1:  Rational r = new Rational(0.0, 0.0);
    r.abs();
2:  Rational n = r.result;
    n.scale(m);
  }
}
```

Figure 9: Rational Number and Client Classes

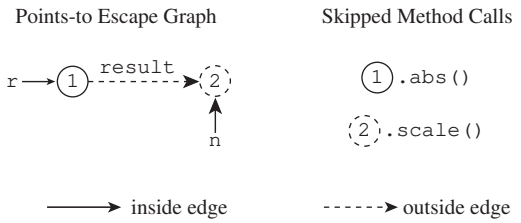

Figure 10: Analysis Result from `evaluate` Method

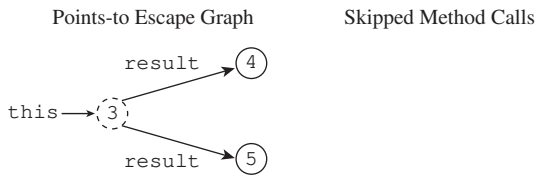

Figure 11: Analysis Result from `abs` Method

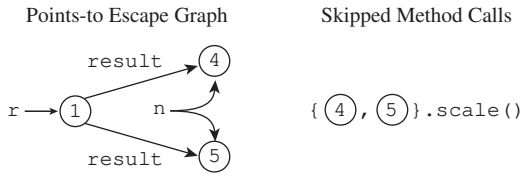

Figure 12: Analysis Result from `evaluate` After Integrating Result from `abs`

ment, in this case the statement `n = r.result` at line 2 in the `evaluate` method.

The analysis policy notices that the target node (node 1) escapes via a call to the `abs` method. It therefore directs the analysis to analyze `abs` and integrate the result into the result from the end of the `evaluate` method. Figure 11 presents the analysis result from the end of the `abs` method. Node 3 represents the receiver object, node 4 represents the object created at line 4 of the `abs` method, and node 5 represents the object created at line 5. The solid edges from node 3 to nodes 4 and 5 are *inside* edges. Inside edges represent references created within the analyzed region of the program, in this case the `abs` method.

The algorithm next integrates this graph into the analysis result from `evaluate`. The goal is to reconstruct the result of the base whole-program analysis. In the base analysis, which does not skip call sites, the analysis of `abs` changes the points-to escape graph at the program point after the call site. These changes in turn affect the analysis of the statements in `evaluate` after the call to `abs`. The incrementalized analysis reconstructs the analysis result as follows. It first determines that node 3 represented node 1 during the analysis of `abs`. It then matches the outside edge against the two inside edges to determine that, during the analysis of the region of `evaluate` after the skipped call to `abs`, the outside edge from node 1 to node 2 represented the inside edges from node 3 to nodes 4 and 5, and that the load node 2 therefore represented nodes 4 and 5. The combined graph therefore contains inside edges from node 1 to nodes 4 and 5. Because node 1 is captured, the analysis removes the outside edge from this node. Finally, the updated analysis replaces the load node 2 in the skipped call site to `scale` with nodes 4 and 5. At this point the analysis has captured node 1 inside the `evaluate` method, enabling the compiler to stack allocate all of the objects created at the corresponding allocation site at line 1 in Figure 9.

3. THE BASE ANALYSIS

The base analysis is a previously published points-to and escape analysis [16]. For completeness, we present the algorithm again here. The algorithm is compositional, analyzing each method once before its callers to extract a single parameterized analysis result that can be specialized for use at different call sites.[1] It therefore analyzes the program in a bottom-up fashion from the leaves of the call graph towards the root. To simplify the presentation we ignore static class variables, exceptions, and return values. Our implemented algorithm correctly handles all of these features.

3.1 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes, which consists of a set N_I of inside nodes and a set N_O of outside nodes. Inside nodes represent objects created inside the currently analyzed region of the program, i.e., inside the current method or one of the analyzed methods that it (transitively) invokes. There is one inside node for each object allocation site; that node represents all objects created at that site. The inside nodes include the set of thread nodes $N_T \subseteq N_I$. Thread nodes represent thread objects, i.e. objects that inherit from `Thread` or implement the `Runnable` interface.

[1]Recursive programs require a fixed-point algorithm that may analyze methods involved in cycles in the call graph multiple times.

The set of parameter nodes $N_P \subseteq N_O$ represents objects passed as parameters into the currently analyzed method. There is one load node $n \in N_L \subseteq N_O$ for each load statement in the program; that node represents all objects whose references are 1) loaded at that statement, and 2) not identified as created inside the currently analyzed region of the program. There is also a set $\mathtt{f} \in \mathtt{F}$ of fields in objects, a set $\mathtt{v} \in \mathtt{V}$ of local or parameter variables, and a set $\mathtt{l} \in \mathtt{L} \subseteq \mathtt{V}$ of local variables.

3.2 Points-To Escape Graphs

A points-to escape graph is a pair $\langle O, I \rangle$, where

- $O \subseteq (N \times \mathtt{F}) \times N_L$ is a set of outside edges. We write an edge $\langle \langle n_1, \mathtt{f} \rangle, n_2 \rangle$ as $n_1 \xrightarrow{\mathtt{f}} n_2$.
- $I \subseteq ((N \times \mathtt{F}) \times N) \cup (\mathtt{V} \times N)$ is a set of inside edges. We write an edge $\langle \mathtt{v}, n \rangle$ as $\mathtt{v} \rightarrow n$ and an edge $\langle \langle n_1, \mathtt{f} \rangle, n_2 \rangle$ as $n_1 \xrightarrow{\mathtt{f}} n_2$.

Inside edges represent references created within the currently analyzed part of the program. Outside edges represent references not identified as created within the currently analyzed part of the program. Outside edges usually represent references created outside the currently analyzed part of the program, but when multiple nodes represent the same object (for example, when a method is invoked with aliased parameters), an outside edge from one node can represent a reference from the object created within the currently analyzed part of the program.

A node escapes if it is reachable in $O \cup I$ from a parameter node or a thread node. We formalize this notion by defining an escape function

$$e_{O,I}(n) = \{n' \in N_T \cup N_P . n \text{ is reachable from } n' \text{ in } O \cup I\}$$

that returns the set of parameter and thread nodes through which n escapes. We define the concepts of escaped and captured nodes as follows:

- escaped$(\langle O, I \rangle, n)$ if $e_{O,I}(n) \neq \emptyset$
- captured$(\langle O, I \rangle, n)$ if $e_{O,I}(n) = \emptyset$

We say that an allocation site escapes or is captured in the context of a given analysis if the corresponding inside node is escaped or captured in the points-to escape graph that the analysis produces.

3.3 Program Representation

The algorithm represents the computation of each method using a control flow graph. We assume the program has been preprocessed so that all statements relevant to the analysis are either a copy statement $\mathtt{l} = \mathtt{v}$, a load statement $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$, a store statement $\mathtt{l}_1.\mathtt{f} = \mathtt{l}_2$, an object allocation statement $\mathtt{l} = \mathtt{new\ cl}$, or a method call statement $\mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$.

3.4 Intraprocedural Analysis

The intraprocedural analysis is a forward dataflow analysis that produces a points-to escape graph for each program point in the method. Each method is analyzed under the assumption that the parameters are *maximally unaliased*, i.e., point to different objects. For a method with formal parameters $\mathtt{v}_0, \ldots, \mathtt{v}_n$, the initial points-to escape graph at the entry point of the method is $\langle \emptyset, \{\langle \mathtt{v}_i, n_{\mathtt{v}_i} \rangle . 1 \leq i \leq n\} \rangle$

where $n_{\mathtt{v}_i}$ is the parameter node for parameter $\mathtt{v}_i$. If the method is invoked in a context where some of the parameters may point to the same object, the interprocedural analysis described below in Section 3.5 merges parameter nodes to conservatively model the effect of the aliasing.

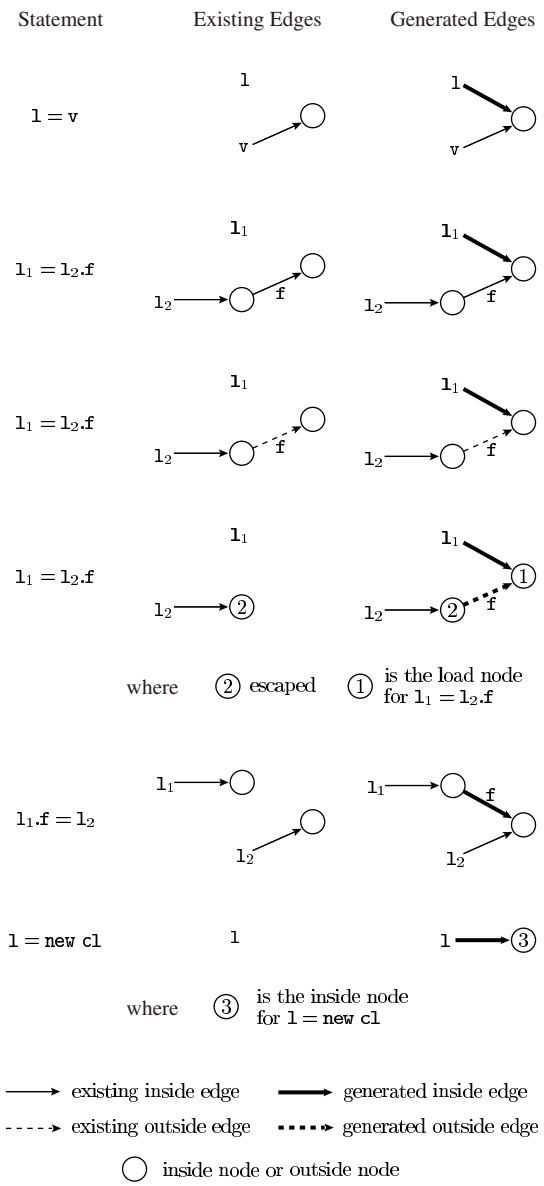

Figure 13: Generated Edges for Basic Statements

The transfer function $\langle O', I' \rangle = [\![\mathtt{st}]\!](\langle O, I \rangle)$ models the effect of each statement `st` on the current points-to escape graph. Figure 13 graphically presents the rules that determine the new graph for each statement. Each row in this figure contains three items: a statement, a graphical representation of existing edges, and a graphical representation of the existing edges plus the new edges that the statement generates. Two of the rows (for statements $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$ and $\mathtt{l} = \mathtt{new\ cl}$) also have a where clause that specifies a set of side conditions. The interpretation of each row is that whenever the points-to escape graph contains the existing edges and the side conditions are satisfied, the transfer function for the statement generates the new edges. Assignments to a variable kill existing edges from that variable; assignments to fields of objects leave existing edges in place. At control-flow merges, the analysis takes the union of the inside and outside edges. At the end of the method, the analysis removes all captured nodes and local or parameter variables from the points-to escape graph.

3.5 Interprocedural Analysis

At each call statement, the interprocedural analysis uses the analysis result from each potentially invoked method to compute a transfer function for the statement. We assume a call site of the form $\mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$, a potentially invoked method `op` with formal parameters $\mathtt{v}_0, \ldots, \mathtt{v}_k$, a points-to escape graph $\langle O_1, I_1 \rangle$ at the program point before the call site, and a graph $\langle O_2, I_2 \rangle$ from the end of `op`.

A map $\mu \subseteq N \times N$ combines the callee graph into the caller graph. The map serves two purposes: 1) it maps each outside node in the callee to the nodes in the caller that it represents during the analysis of the callee, and 2) it maps each node in the callee to itself if that node should be present in the combined graph. We use the notation $\mu(n) = \{n' . \langle n, n' \rangle \in \mu\}$ and $n_1 \xrightarrow{\mu} n_2$ for $n_2 \in \mu(n_1)$.

The interprocedural mapping algorithm $\langle \langle O, I \rangle, \mu \rangle = \mathrm{map}(\langle O_1, I_1 \rangle, \langle O_2, I_2 \rangle, \hat{\mu})$ starts with the points-to escape graph $\langle O_1, I_1 \rangle$ from the caller, the graph $\langle O_2, I_2 \rangle$ from the callee, and an initial parameter map

$$\hat{\mu}(n) = \begin{cases} I_1(\mathtt{l}_i) & \text{if } \{n\} = I_2(\mathtt{v}_i) \\ \emptyset & \text{otherwise} \end{cases}$$

that maps each parameter node from the callee to the nodes that represent the corresponding actual parameters at the call site. It produces the new mapped edges from the callee $\langle O, I \rangle$ and the new map μ.

Figure 14 presents the constraints that define the new edges $\langle O, I \rangle$ and new map μ. Constraint 1 initializes the map μ to the initial parameter map $\hat{\mu}$. Constraint 2 extends μ, matching outside edges from the callee against edges from the caller to ensure that μ maps each outside node from the callee to the corresponding nodes in the caller that it represents during the analysis of the callee. Constraint 3 extends μ to model situations in which aliasing in the caller causes an outside node from the callee to represent other callee nodes during the analysis of the callee. Constraints 4 and 5 complete the map by computing which nodes from the callee should be present in the caller and mapping these nodes to themselves. Constraints 6 and 7 use the map to translate inside and outside edges from the callee into the caller. The new graph at the program point after the call site is $\langle I_1 \cup I, O_1 \cup O \rangle$.

Because of dynamic dispatch, a single call site may invoke several different methods. The transfer function therefore merges the points-to escape graphs from the analysis of all potentially invoked methods to derive the new graph at the point after the call site. The current implementation obtains this call graph information using a variant of a cartesian product type analysis [1], but it can use any conservative approximation to the dynamic call graph.

3.6 Merge Optimization

As presented so far, the analysis may generate points-to escape graphs $\langle O, I \rangle$ in which a node n may have multiple distinct outside edges $n \xrightarrow{\mathtt{f}} n_1, \ldots, n \xrightarrow{\mathtt{f}} n_k \in O$. We eliminate this inefficiency by merging the load nodes $n_1, \ldots, n_k$.

$$\hat{\mu}(n) \subseteq \mu(n) \tag{1}$$

$$\frac{n_1 \xrightarrow{\mathtt{f}} n_2 \in O_2, n_3 \xrightarrow{\mathtt{f}} n_4 \in O_1 \cup I_1, n_1 \xrightarrow{\mu} n_3}{n_2 \xrightarrow{\mu} n_4} \tag{2}$$

$$\frac{\begin{array}{c} n_1 \xrightarrow{\mu} n_3, n_2 \xrightarrow{\mu} n_3, n_1 \neq n_2, \\ n_1 \xrightarrow{\mathtt{f}} n_4 \in O_2, n_2 \xrightarrow{\mathtt{f}} n_5 \in O_2 \cup I_2 \end{array}}{\mu(n_4) \subseteq \mu(n_5)} \tag{3}$$

$$\frac{n_1 \xrightarrow{\mathtt{f}} n_2 \in I_2, n_1 \xrightarrow{\mu} n, n_2 \in N_I}{n_2 \xrightarrow{\mu} n_2} \tag{4}$$

$$\frac{n_1 \xrightarrow{\mathtt{f}} n_2 \in O_2, n_1 \xrightarrow{\mu} n, \mathrm{escaped}(\langle O, I \rangle, n)}{n_2 \xrightarrow{\mu} n_2} \tag{5}$$

$$\frac{n_1 \xrightarrow{\mathtt{f}} n_2 \in I_2}{(\mu(n_1) \times \{\mathtt{f}\}) \times \mu(n_2) \subseteq I} \tag{6}$$

$$\frac{n_1 \xrightarrow{\mathtt{f}} n_2 \in O_2, n_2 \xrightarrow{\mu} n_2}{(\mu(n_1) \times \{\mathtt{f}\}) \times \{n_2\} \subseteq O} \tag{7}$$

Figure 14: Constraints for Interprocedural Analysis

With this optimization, a single load node may be associated with multiple load statements. The load node generated from the merge of k load nodes $n_1, \ldots, n_k$ is associated with all of the statements of $n_1, \ldots, n_k$.

4. THE INCREMENTALIZED ANALYSIS

We next describe how to *incrementalize* the base algorithm — how to enhance the algorithm so that it can skip the analysis of call sites while maintaining enough information to reconstruct the result of analyzing the invoked methods should the analysis policy direct the analysis to do so. The first step is to record the set S of skipped call sites. For each skipped call site s, the analysis records the invoked method $\mathtt{op}_s$ and the initial parameter map $\hat{\mu}_s$ that the base algorithm would compute at that call site. To simplify the presentation, we assume that each skipped call site is 1) executed at most once, and 2) invokes a single method. Section 4.8 discusses how we eliminate these restrictions in our implemented algorithm.

The next step is to define an updated escape function $e_{S,O,I}$ that determines how objects escape the currently analyzed region of the program via skipped call sites:

$$\begin{array}{l} e_{S,O,I}(n) = \{s \in S. \exists n_1 \in N_P. n_1 \xrightarrow{\hat{\mu}_s} n_2 \text{ and} \\ \qquad\qquad n \text{ is reachable from } n_2 \text{ in } O \cup I\} \cup e_{O,I}(n) \end{array}$$

We adapt the interprocedural mapping algorithm from Section 3.5 to use this updated escape function. By definition, n escapes through a call site s if $s \in e_{S,O,I}(n)$.

A key complication is preserving flow sensitivity with respect to previously skipped call sites during the integration of analysis results from those sites. For optimization purposes, the compiler works with the analysis result from the end of the method. But the skipped call sites occur at various program points inside the method. We therefore augment the points-to escape graphs from the base analysis with several orders, which record ordering information between edges in the points-to escape graph and skipped call sites:

- $\omega \subseteq S \times ((N \times \{\mathtt{f}\}) \times N_L)$. For each call site s, $\omega(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2.\langle s, n_1 \xrightarrow{\mathtt{f}} n_2\rangle \in \omega\}$ is the set of outside edges that the analysis generates before it skips s.
- $\iota \subseteq S \times ((N \times \{\mathtt{f}\}) \times N)$. For each call site s, $\iota(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2.\langle s, n_1 \xrightarrow{\mathtt{f}} n_2\rangle \in \iota\}$ is the set of inside edges that the analysis generates before it skips s.
- $\tau \subseteq S \times ((N \times \{\mathtt{f}\}) \times N_L)$. For each call site s, $\tau(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2.\langle s, n_1 \xrightarrow{\mathtt{f}} n_2\rangle \in \tau\}$ is the set of outside edges that the analysis generates after it skips s.
- $\nu \subseteq S \times ((N \times \{\mathtt{f}\}) \times N)$. For each call site s, $\nu(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2.\langle s, n_1 \xrightarrow{\mathtt{f}} n_2\rangle \in \nu\}$ is the set of inside edges that the analysis generates after it skips s.
- $\beta \subseteq S \times S$. For each call site s, $\beta(s) = \{s'.\langle s, s'\rangle \in \beta\}$ is the set of call sites that the analysis skips before skipping s.
- $\alpha \subseteq S \times S$. For each call site s, $\alpha(s) = \{s'.\langle s, s'\rangle \in \alpha\}$ is the set of call sites that the analysis skips after skipping s.

The incrementalized analysis works with augmented points-to escape graphs of the form $\langle O, I, S, \omega, \iota, \tau, \nu, \beta, \alpha\rangle$. Note that because β and α are inverses,[2] the analysis does not need to represent both explicitly. It is of course possible to use any conservative approximation of ω, ι, τ, ν, β and α; an especially simple approach uses $\omega(s) = \tau(s) = O$, $\iota(s) = \nu(s) = I$, and $\beta(s) = \alpha(s) = S$.

We next discuss how the analysis uses these additional components during the incremental analysis of a call site. We assume a current augmented points-to escape graph $\langle O_1, I_1, S_1, \omega_1, \iota_1, \tau_1, \nu_1, \beta_1, \alpha_1\rangle$, a call site $s \in S_1$ with invoked operation $\mathtt{op}_s$, and an augmented points-to escape graph $\langle O_2, I_2, S_2, \omega_2, \iota_2, \tau_2, \nu_2, \beta_2, \alpha_2\rangle$ from the end of $\mathtt{op}_s$.

4.1 Matched Edges

In the base algorithm, the analysis of a call site matches outside edges from the analyzed method against existing edges in the points-to escape graph from the program point before the site. By the time the algorithm has propagated the graph to the end of the method, it may contain additional edges generated by the analysis of statements that execute after the call site. When the incrementalized algorithm integrates the analysis result from a skipped call site, it matches outside edges from the invoked method against only those edges that were present in the points-to escape graph at the program point before the call site. $\omega(s)$ and $\iota(s)$ provide just those edges. The algorithm therefore computes

$$\langle O, I, \mu\rangle = \mathrm{map}(\langle \omega_1(s), \iota_1(s)\rangle, \langle O_2, I_2\rangle, \hat{\mu}_s)$$

where O and I are the new sets of edges that the analysis of the callee adds to the caller graph.

[2] Under the interpretation $\beta^{-1} = \{\langle s_1, s_2\rangle.\langle s_2, s1\rangle \in \beta\}$ and $\alpha^{-1} = \{\langle s_1, s_2\rangle.\langle s_2, s1\rangle \in \alpha\}$, $\beta = \alpha^{-1}$ and $\beta^{-1} = \alpha$.

4.2 Propagated Edges

In the base algorithm, the transfer function for an analyzed call site may add new edges to the points-to graph from before the site. These new edges create effects that propagate through the analysis of subsequent statements. Specifically, the analysis of these subsequent statements may read the new edges, then generate additional edges involving the newly referenced nodes. In the points-to graph from the incrementalized algorithm, the edges from the invoked method will not be present if the analysis skips the call site. But these missing edges must come (directly or indirectly) from nodes that escape into the skipped call site. In the points-to graphs from the caller, these missing edges are represented by outside edges that are generated by the analysis of subsequent statements. The analysis can therefore use $\tau_1(s)$ and $\nu_1(s)$ to reconstruct the propagated effect of analyzing the skipped method. It computes

$$\langle O', I', \mu' \rangle = \text{map}(\langle O, I \rangle, \langle \tau_1(s), \nu_1(s) \rangle, \{\langle n, n \rangle . n \in N\})$$

where O' and I' are the new sets of edges that come from the interaction of the analysis of the skipped method and subsequent statements, and μ' maps each outside node from the caller to the nodes from the callee that it represents during the analysis from the program point after the skipped call site to the end of the method. Note that this algorithm generates all of the new edges that a complete reanalysis would generate. But it generates the edges incrementally without reanalyzing the code.

4.3 Skipped Call Sites from the Caller

In the base algorithm, the analysis of one call site may affect the initial parameter map for subsequent call sites. Specifically, the analysis of a site may cause the formal parameter nodes at subsequent sites to be mapped to additional nodes in the graph from the caller.

For each skipped call site, the incrementalized algorithm records the parameter map that the base algorithm would have used at that site. When the incrementalized algorithm integrates an analysis result from a previously skipped site, it must update the recorded parameter maps for subsequent skipped sites. At each of these sites, outside nodes represent the additional nodes that the analysis of the previously skipped site may add to the map. And the map μ' records how each of these outside nodes should be mapped. For each subsequent site $s' \in \alpha_1(s)$, the algorithm composes the site's current recorded parameter map $\hat{\mu}_{s'}$ with μ' to obtain its new recorded parameter map $\mu' \circ \hat{\mu}_{s'}$.

4.4 Skipped Call Sites from the Callee

The new set of skipped call sites $S' = (S_1 \cup S_2)$ contains the set of skipped call sites S_2 from the callee. When it maps the callee graph into the caller graph, the analysis updates the recorded parameter maps for the skipped call sites in S_2. For each site $s' \in S_2$, the analysis simply composes the site's current map $\hat{\mu}_{s'}$ with the map μ to obtain the new recorded parameter map $\mu \circ \hat{\mu}_{s'}$ for s'.

4.5 New Orders

The analysis constructs the new orders by integrating the orders from the caller and callee into the new analysis result and extending the orders for s to the mapped edges and skipped call sites from the callee. So, for example, the new order between outside edges and subsequent call sites (ω') consists of the order from the caller (ω_1), the mapped order from the callee ($\omega_2[\mu]$), the order from s extended to the skipped call sites from the callee ($S_2 \times \omega_1(s)$), and the outside edges from the callee ordered with respect to the call sites after s ($\alpha_1(s) \times O$):

$$\begin{aligned}
\omega' &= \omega_1 \cup \omega_2[\mu] \cup (S_2 \times \omega_1(s)) \cup (\alpha_1(s) \times O) \\
\iota' &= \iota_1 \cup \iota_2[\mu] \cup (S_2 \times \iota_1(s)) \cup (\alpha_1(s) \times I) \\
\tau' &= \tau_1 \cup \tau_2[\mu] \cup (S_2 \times \tau_1(s)) \cup (\beta_1(s) \times O) \\
\nu' &= \nu_1 \cup \nu_2[\mu] \cup (S_2 \times \nu_1(s)) \cup (\beta_1(s) \times I) \\
\beta' &= \beta_1 \cup \beta_2 \cup (S_2 \times \beta_1(s)) \cup (\alpha_1(s) \times S_2) \\
\alpha' &= \alpha_1 \cup \alpha_2 \cup (S_2 \times \alpha_1(s)) \cup (\beta_1(s) \times S_2)
\end{aligned}$$

Here $\omega[\mu]$ is the order ω under the map μ, i.e., $\omega[\mu] = \{\langle s, n_1' \xrightarrow{f} n_2' \rangle . \langle s, n_1 \xrightarrow{f} n_2 \rangle \in \omega, n_1 \xrightarrow{\mu} n_1', \text{ and } n_2 \xrightarrow{\mu} n_2'\}$, and similarly for ι, τ, and ν.

4.6 Cleanup

At this point the algorithm can compute a new graph $\langle O_1 \cup O \cup O', I_1 \cup I \cup I', S', \omega', \iota', \tau', \nu', \beta', \alpha' \rangle$ that reflects the integration of the analysis of s into the previous analysis result $\langle O_1, I_1, S_1, \omega_1, \iota_1, \tau_1, \nu_1, \beta_1, \alpha_1 \rangle$. The final step is to remove s from all components of the new graph and to remove all outside edges from captured nodes.

4.7 Updated Intraprocedural Analysis

The transfer function for a skipped call site s performs the following additional tasks:

- Record the initial parameter map $\hat{\mu}_s$ that the base algorithm would use when it analyzed the site.
- Update ω to include $\{s\} \times O$, update ι to include $\{s\} \times I$, update α to contain $S \times \{s\}$, and update β to contain $\{s\} \times S$.
- Update S to include the skipped call site s.

Whenever a load statement generates a new outside edge $n_1 \xrightarrow{f} n_2$, the transfer function updates τ to include $S \times \{n_1 \xrightarrow{f} n_2\}$. Whenever a store statement generates a new inside edge $n_1 \xrightarrow{f} n_2$, the transfer function updates ν to include $S \times \{n_1 \xrightarrow{f} n_2\}$.

Finally, the incrementalized algorithm extends the confluence operator to merge the additional components. For each additional component (including the recorded parameter maps μ_s), the confluence operator is set union.

4.8 Extensions

So far, we have assumed that each skipped call site is executed at most once and invokes a single method. We next discuss how our implemented algorithm eliminates these restrictions. To handle dynamic dispatch, we compute the graph for all of the possible methods that the call site may invoke, then merge these graphs to obtain the new graph.

We also extend the abstraction to handle skipped call sites that are in loops or are invoked via multiple paths in the control flow graph. We maintain a multiplicity flag for each call site specifying whether the call site may be executed multiple times:

- The transfer function for a skipped call site s checks to see if the site is already in the set of skipped sites S. If so, it sets the multiplicity flag to indicate that s may be invoked multiple times. It also takes the union of

the site's current recorded parameter map $\hat{\mu}_s$ and the parameter map $\hat{\mu}$ from the transfer function to obtain the site's new recorded parameter map $\hat{\mu}_s \cup \hat{\mu}$.

- The algorithm that integrates analysis results from previously skipped call sites performs a similar set of operations to maintain the recorded parameter maps and multiplicity flags for call sites that may be present in the analysis results from both the callee and the caller. If the skipped call site may be executed multiple times, the analysis uses a fixed-point algorithm when it integrates the analysis result from the skipped call site. This algorithm models the effect of executing the site multiple times.

4.9 Recursion

The base analysis uses a fixed-point algorithm to ensure that it terminates in the presence of recursion. It is possible to use a similar approach in the incrementalized algorithm. Our implemented algorithm, however, does not check for recursion as it explores the call graph. If a node escapes into a recursive method, the analysis may, in principle, never terminate. In practice, the algorithm relies on the analysis policy to react to the expansion of the analyzed region by directing analysis resources to other allocation sites.

4.10 Incomplete Call Graphs

Our algorithm deals with incomplete call graphs as follows. If it is unable to locate all of the potential callers of a given method, it simply analyzes those it is able to locate. If it is unable to locate all potential callees at a given call site, it simply considers all nodes that escape into the site as permanently escaped.

5. ANALYSIS POLICY

The goal of the analysis policy is to find and analyze allocation sites that can be captured quickly and have a large optimization payoff. Conceptually, the policy uses the following basic approach. It estimates the payoff for capturing an allocation site as the number of objects allocated at that site in a previous profiling run. It uses empirical data and the current analysis result for the site to estimate the likelihood that it will ever be able to capture the site, and, assuming that it is able to capture the site, the amount of time required to do so. It then uses these estimates to calculate an estimated marginal return for each unit of analysis time invested in each site.

At each analysis step, the policy is faced with a set of partially analyzed sites that it can invest in. The policy simply chooses the site with the best estimated marginal return, and invests a (configurable) unit of analysis time in that site. During this time, the algorithm repeatedly selects one of the skipped call sites through which the allocation site escapes, analyzes the methods potentially invoked at that site (reusing the cached results if they are available), and integrates the results from these methods into the current result for the allocation site. If these analyses capture the site, the policy moves on to the site with the next best estimated marginal return. Otherwise, when the time expires, the policy recomputes the site's estimated marginal return in light of the additional information it has gained during the analysis, and once again invests in the (potentially different) site with the current best estimated marginal return.

5.1 Stack Allocation

The compiler applies two potential stack allocation optimizations depending on where an allocation site is captured:

- **Stack Allocate:** If the site is captured in the method that contains it, the compiler generates code to allocate all objects created at that site in the activation record of the containing method.

- **Inline and Stack Allocate:** If the site is captured in a direct caller of the method containing the site, the compiler first inlines the method into the caller. After inlining, the caller contains the site, and the generated code allocates all objects created at that site in the activation record of the caller.

The current analysis policy assumes that the compiler is 1) unable to inline a method if, because of dynamic dispatch, the corresponding call site may invoke multiple methods, and 2) unwilling to enable additional optimizations by further inlining the callers of the method containing the allocation site into their callers. It is, of course, possible to relax these assumptions to support more sophisticated inlining and/or specialization strategies.

Inlining complicates the conceptual analysis policy described above. Because each call site provides a distinct analysis context, the same allocation site may have different analysis characteristics and outcomes when its enclosing method is inlined at different call sites. The policy therefore treats each distinct combination of call site and allocation site as its own separate analysis opportunity.

5.2 Analysis Opportunities

The policy represents an opportunity to capture an allocation site a in its enclosing method `op` as $\langle a, \texttt{op}, G, p, c, d, m\rangle$, where G is the current augmented points-to escape graph for the site, p is the estimated payoff for capturing the site, c is the count of the number of skipped call sites in G through which a escapes, d is the method call depth of the analyzed region represented by G, and m is the mean cost of the call site analyses performed so far on behalf of this analysis opportunity. Note that a, `op`, and G are used to perform the incremental analysis, while p, c, d, and m are used to estimate the marginal return. Opportunities to capture an allocation site a in the caller `op` of its enclosing method have the form $\langle a, \texttt{op}, s, G, p, c, d, m\rangle$, where s is the call site in `op` that invokes the method containing a, and the remainder of the fields have the same meaning as before.

Figure 15 presents the state-transition diagram for analysis opportunities. Each analysis opportunity can be in one of the states of the diagram; the transitions correspond to state changes that take place during the analysis of the opportunity. The states have the following meanings:

- **Unanalyzed:** No analysis done on the opportunity.

- **Escapes Below Enclosing Method:** The opportunity's allocation site escapes into one or more skipped call sites, but does not (currently) escape to the caller of the enclosing method. The opportunity is of the form $\langle a, \texttt{op}, G, p, c, d, m\rangle$.

- **Escapes Below Caller of Enclosing Method:** The opportunity's site escapes to the caller of its enclosing method, but does not (currently) escape from this

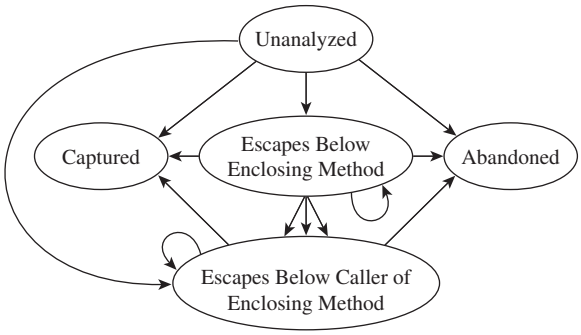

Figure 15: State-Transition Diagram for Analysis Opportunities

caller. The site may also escape into one or more skipped call sites. The opportunity is of the form $\langle a, \mathtt{op}, s, G, p, c, d, m\rangle$.

- **Captured:** The opportunity's site is captured.
- **Abandoned:** The policy has permanently abandoned the analysis of the opportunity, either because its allocation site permanently escapes via a static class variable or thread, because the site escapes to the caller of the caller of its enclosing method (and is therefore unoptimizable), or because the site escapes to the caller of its enclosing method and (because of dynamic dispatch) the compiler is unable to inline the enclosing method into the caller.

In Figure 15 there are multiple transitions from the Escapes Below Enclosing Method state to the Escapes Below Caller of Enclosing Method state. These transitions indicate that one Escapes Below Enclosing Method opportunity may generate multiple new Escapes Below Caller of Enclosing Method opportunities — one new opportunity for each potential call site that invokes the enclosing method from the old opportunity.

When an analysis opportunity enters the Escapes Below Caller of Enclosing Method state, the first analysis action is to integrate the augmented points-to escape graph from the enclosing method into the graph from the caller of the enclosing method.

5.3 Estimated Marginal Returns

If the opportunity is Unanalyzed, the estimated marginal return is $(\xi \cdot p)/\sigma$, where ξ is the probability of capturing an allocation site given no analysis information about the site, p is the payoff of capturing the site, and, assuming the analysis eventually captures the site, σ is the expected analysis time required to do so.

If the opportunity is in the state Escapes Below Enclosing Method, the estimated marginal return is $(\xi_1(d) \cdot p)/(c \cdot m)$. Here $\xi_1(d)$ is the conditional probability of capturing an allocation site given that the algorithm has explored a region of call depth d below the method containing the site, the algorithm has not (yet) captured the site, and the site has not escaped (so far) to the caller of its enclosing method. If the opportunity is in the state Escapes Below Caller of Enclosing Method, the estimated marginal return is $(\xi_2(d) \cdot p)/(c \cdot m)$. Here $\xi_2(d)$ has the same meaning as $\xi_1(d)$, except that the assumption is that the site has escaped to the caller of its enclosing method, but not (so far) to the caller of the caller of its enclosing method.

We obtain the capture probability functions ξ, ξ_1, and ξ_2 empirically by preanalyzing all of the executed allocation sites in some sample programs and collecting data that allows us to compute these functions. For Escapes Below Enclosing Method opportunities, the estimated payoff p is the number of objects allocated at the opportunity's allocation site a during a profiling run. For Escapes Below Caller of Enclosing Method opportunities, the estimated payoff is the number of objects allocated at the opportunity's allocation site a when the allocator is invoked from the opportunity's call site s.

When an analysis opportunity changes state or increases its method call depth, its estimated marginal return may change significantly. The policy therefore recomputes the opportunity's return whenever one of these events happens. If the best opportunity changes because of this recomputation, the policy redirects the analysis to work on the new best opportunity.

5.4 Termination

In principle, the policy can continue the analysis indefinitely as it invests in ever less profitable opportunities. In practice, it is important to terminate the analysis when the prospective returns become small compared to the analysis time required to realize them. We say that the analysis has *decided* an object if that object's opportunity is in the Captured or Abandoned state. The payoffs p in the analysis opportunities enable the policy to compute the current number of decided and undecided objects.

Two factors contribute to our termination policy: the percentage of undecided objects (this percentage indicates the maximum potential payoff from continuing the analysis), and the rate at which the analysis has recently been deciding objects. The results in Section 6 are from analyses terminated when the percentage of decided objects rises above 90% and the decision rate for the last quarter of the analysis drops below 1 percent per second, with a cutoff of 75 seconds of total analysis time.

We anticipate the development of a variety of termination policies to fit the particular needs of different compilers. A dynamic compiler, for example, could accumulate an analysis budget as a percentage of the time spent executing the application — the longer the application ran, the more time the policy would be authorized to invest analyzing it. The accumulation rate would determine the maximum amortized analysis overhead.

6. EXPERIMENTAL RESULTS

We have implemented our analysis and the stack allocation optimization in the MIT Flex compiler, an ahead-of-time compiler written in Java for Java.[3] We ran the experiments on an 800 MHz Pentium III PC with 768Mbytes of memory running Linux Version 2.2.18. We ran the compiler using the Java Hotspot Client VM version 1.3.0 for Linux. The compiler generates portable C code, which we compile to an executable using gcc. The generated code manages the heap using the Boehm-Demers-Weiser conservative garbage collector [4] and uses `alloca` for stack allocation.

[3]The compiler is available at www.flexc.lcs.mit.edu.

6.1 Benchmark Programs

Our benchmark programs include two multithreaded scientific computations (Barnes and Water), Jlex, and several Spec benchmarks (Db, Compress, and Raytrace). Barnes and Water are well-known benchmarks in the parallel computing community; our Java versions were derived from versions in the SPLASH-2 benchmark suite [17]. Figure 16 presents the compile and whole-program analysis times for the applications.

Application	Compile Time Without Analysis	Whole-Program Analysis Time
Barnes	89.7	34.3
Water	91.1	38.2
Jlex	119.5	222.8
Db	93.6	126.6
Raytrace	118.4	102.2
Compress	219.6	645.1

Figure 16: Compile and Whole-Program Analysis Times (seconds)

6.2 Marginal Returns and Profiling Information

We derived the estimated capture probability functions ξ, ξ_1, and ξ_2 from an instrumented analysis of all of the executed object allocation sites in Barnes, Water, Db, and Raytrace. Figure 17 presents the capture probabilities $\xi_1(d)$ and $\xi_2(d)$ as a function of the call depth d; ξ is .33.

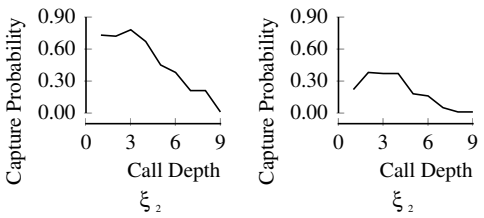

Figure 17: Capture Probability Functions

To compute the estimated marginal returns and implement the termination policy, the analysis policy needs an estimated optimization payoff for each allocation site. We obtain these payoffs as the number of objects allocated at each site during a training run on a small training input. The presented execution and analysis statistics are for executions on larger production inputs.

6.3 Analysis Payoffs and Statistics

Figure 18 presents analysis statistics from the incrementalized analysis. We present the number of captured allocation sites as the sum of two counts. The first count is the number of sites captured in the enclosing method; the second is the number captured in the caller of the enclosing method. Fractional counts indicate allocation sites that were captured in some but not all callers of the enclosing method. In Db, for example, one of the allocation sites is captured in two of the eight callers of its enclosing method. The Undecided Allocation Sites column counts the number of allocation sites in which the policy invested some resources, but did not determine whether it could stack allocate the corresponding objects or not. The Analyzed Call Sites, Total Call Sites, Analyzed Methods, and Total Methods columns show that the policy analyzes a small fraction of the total program.

The graphs in Figure 19 present three curves for each application. The horizontal dotted line indicates the percentage of objects that the whole-program analysis allocates on the stack. The dashed curve plots the percentage of decided objects (objects whose analysis opportunities are either Captured or Abandoned) as a function of the analysis time. The solid curve plots the percentage of objects allocated on the stack. For Barnes, Jlex, and Db, the incrementalized analysis captures virtually the same number of objects as the whole-program analysis, but spends a very small fraction of the whole-program analysis time to do so. Incrementalization provides less of a benefit for Water because two large methods account for a much of the analysis time of both analyses. For Raytrace and Compress, a bug in the 1.3 JVM forced us to run the incrementalized analysis, but not the whole-program analysis, on the 1.2 JVM. Our experience with the other applications indicates that both analyses run between five and six times faster on the 1.3 JVM than on the 1.2 JVM.

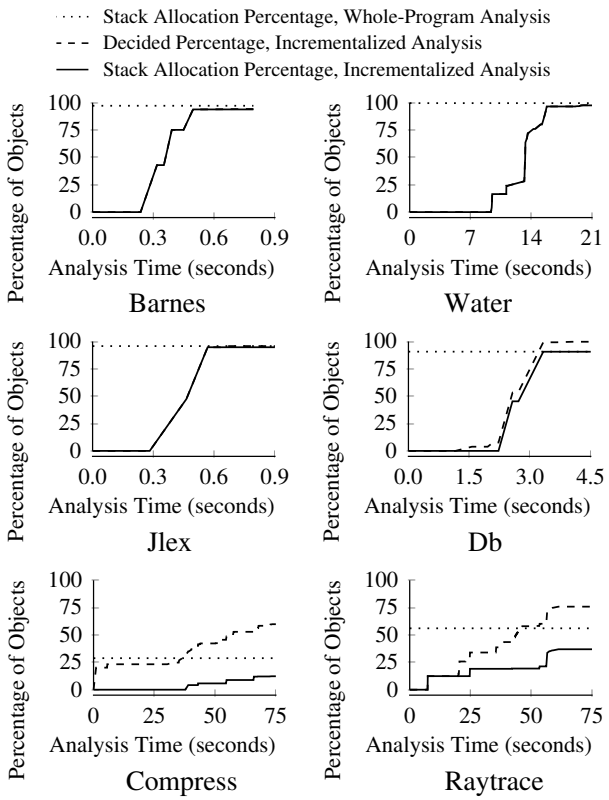

Figure 19: Analysis Time Payoffs

6.4 Application Execution Statistics

Figure 20 presents the total amount of memory that the applications allocate in the heap. Almost all of the allocated memory in Barnes and Water is devoted to temporary arrays that hold the results of intermediate computations. The C++ version of these applications allocates these arrays on the stack; our analysis restores this allocation strategy in the Java version. Most of the memory in Jlex is devoted to temporary iterators, which are stack allocated after inlining. Note the anomaly in Db and Compress: many objects are allocated on the stack, but the heap allocated objects are much bigger than the stack allocated objects.

	Analysis Time (seconds)	Captured Allocation Sites	Abandoned Allocation Sites	Undecided Allocation Sites	Total Allocation Sites	Analyzed Call Sites	Total Call Sites	Analyzed Methods	Total Methods
Barnes	0.8	3+0	0	2	736	18	1675	13	512
Water	21.7	33+0	4	33	748	94	1799	33	481
Jlex	0.9	0+2	1	2	1054	27	2879	12	569
Db	4.5	1+0.25	4	1.75	1118	54	2444	25	631
Raytrace	76.3	8+0.37	20.63	54	1067	271	3109	64	699
Compress	79.5	4+0.33	4	19.66	1354	111	4084	40	808

Figure 18: Analysis Statistics from Incrementalized Analysis

Figure 21 presents the execution times. The optimizations provide a significant performance benefit for Barnes and Water and some benefit for Jlex and Db. Without stack allocation, Barnes and Water interact poorly with the conservative garbage collector. We expect that a precise garbage collector would reduce the performance difference between the versions with and without stack allocation.

Application	No Analysis	Incrementalized Analysis	Whole-Program Analysis
Barnes	36.0	3.2	2.0
Water	190.2	2.2	0.6
Jlex	40.8	3.1	2.5
Db	77.6	31.2	31.2
Raytrace	13.4	9.0	6.7
Compress	110.1	110.1	110.1

Figure 20: Allocated Heap Memory (Mbytes)

Application	No Analysis	Incrementalized Analysis	Whole-Program Analysis
Barnes	33.4	22.7	24.0
Water	18.8	11.2	10.7
Jlex	5.5	5.0	4.7
Db	103.8	104.0	101.3
Raytrace	3.0	2.9	2.9
Compress	44.9	44.8	45.1

Figure 21: Execution Times (seconds)

7. RELATED WORK

We first address related work in escape analysis, focusing on the prospects for incrementalizing existing algorithms. We then discuss several interprocedural analyses (demand-driven analysis, fragment analysis, and incremental analysis) that are designed to analyze part, but not all, of the program.

7.1 Escape Analysis

Many other researchers have developed escape analyses for Java [16, 7, 14, 3, 5]. These analyses have been presented as whole-program analyses, although many contain elements that make them amenable to incrementalization. All of the analyses listed above except the last [5] analyze methods independently of their callers, generating a summary that can be specialized for use at each call site. Unlike our base analysis [16], these analyses are not designed to skip call sites. But we believe it would be relatively straightforward to augment them to do so. With this extension in place, the remaining question is incrementalization. For flow-sensitive analyses [16, 7], the incrementalized algorithm must record information about the ordering of skipped call sites relative to the rest of the analysis information if it is to preserve the precision of the base whole-program analysis with respect to skipped call sites. Flow-insensitive analyses [14, 3], can ignore this ordering information and should therefore be able to use an extended abstraction that records only the mapping information for skipped call sites. In this sense flow-insensitive analyses should be, in general, simpler to incrementalize than flow-sensitive analyses.

Escape analyses have typically been used for stack allocation and synchronization elimination. Our results show that analyzing a local region around each allocation site works well for stack allocation, presumably because stack allocation ties object lifetimes to the lifetimes of the capturing methods. But for synchronization elimination, a whole-program analysis may deliver significant additional optimization opportunities. For example, Ruf's synchronization elimination analysis determines which threads may synchronize on which objects [14]. In many cases, the analysis is able to determine that only one thread synchronizes on a given object, even though the object may be accessible to multiple threads or even, via a static class variable, to all threads. Exploiting this global information significantly improves the ability of the compiler to eliminate superfluous synchronization operations, especially for single threaded programs. We do not see how an incrementalized analysis could extract this kind of global information without scanning all of the code in each thread.

7.2 Demand-Driven Analysis

Demand-driven algorithms analyze only those parts of the program required to compute an analysis fact at a subset of the program points or to answer a given query [2, 10, 8, 11]. Like demand-driven analyses, our analysis does not analyze those parts of the program that do not affect the desired analysis results. Our approach differs in that it is designed to temporarily skip parts of the program even if the skipped parts potentially affect the analysis result. This approach works for its intended application (stack allocation) because it enables the analysis to choose from a set of potential optimization opportunities, some or all of which it is willing to forgo if the analysis cost is too high. In this context, avoiding excessively expensive, even if ultimately successful, analyses is as important as analyzing only those parts of the program required to obtain a specific result. Because our analysis can skip call sites, it can incrementally invest in multiple optimization opportunities, use the acquired information to improve its estimates of the marginal return of each opportunity, then dynamically redirect analysis resources to the currently most promising opportunities. In practice, this approach enables our analysis policy to quickly discover and exploit the best opportunities while avoiding opportunities that provide little or no optimization payoff.

7.3 Fragment and Incremental Analysis

Fragment analysis is designed to analyze a predetermined part of the program [12, 13]. The analysis either extracts a result that is valid for all possible contexts in which the fragment may be placed [12], or is designed to analyze the fragment in the context of a whole-program analysis result from a less expensive algorithm [13]. A similar effect may be obtained by explicitly specifying the analysis results for missing parts of the program [9, 15]. Our approach differs in that it monitors the analysis results to dynamically determine which parts of the program it should analyze to obtain the best optimization outcome.

Incremental algorithms update an existing analysis result to reflect the effect of program changes [18]. Our algorithm, in contrast, analyzes part of the program assuming no previous analysis results.

8. CONCLUSION

This paper presents a new incrementalized pointer and escape analysis. Instead of analyzing the whole program, the analysis executes under the direction of an analysis policy. The policy continually monitors the analysis results to direct the incremental analysis of those parts of the program that offer the best marginal return on the invested analysis resources. Our experimental results show that our analysis, when used for stack allocation, usually delivers almost all of the benefit of the whole-program analysis at a fraction of the cost. And because it analyzes only a local region of the program surrounding each allocation site, it scales to handle programs of arbitrary size.

9. ACKNOWLEDGEMENTS

The research presented in this paper would have been impossible without the assistance of C. Scott Ananian, Brian Demsky, and Alexandru Salcianu. Scott and Brian provided invaluable assistance with the Flex compiler infrastructure. In particular, Brian developed the profiling package on very short notice. Alex provided invaluable assistance with the implementation of the base algorithm. We thank David Karger and Ron Rivest for interesting discussions regarding potential analysis policies, and the anonymous reviewers for their helpful and exceptionally competent feedback.

10. REFERENCES

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, Aug. 1995.

[2] G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proceedings of the 1999 International Conference on Software Maintenance*, Oxford, UK, Aug. 1999.

[3] B. Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[4] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, Sept. 1988.

[5] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[6] G. Bollella et al. *The Real-Time Specification for Java*. Addison-Wesley, Reading, Mass., 2000.

[7] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[8] E. Duesterwald, R. Gupta, and M. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, Nov. 1997.

[9] S. Guyer and C. Lin. Optimizing the use of high performance libraries. In *Proceedings of the Thirteenth Workshop on Languages and Compilers for Parallel Computing*, Yorktown Heights, NY, Aug. 2000.

[10] Y. Lin and D. Padua. Demand-driven interprocedural array property analysis. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, La Jolla, CA, Aug. 1999.

[11] T. Reps, S. Horowitz, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering*, Oct. 1995.

[12] A. Rountev and B. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of CC 2001: International Conference on Compiler Construction*, Genoa, Italy, Apr. 2001.