Compositional Pointer and Escape Analysis for Multithreaded Java Programs

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

rinard@lcs.mit.edu

John Whaley
IBM Tokyo Research Laboratory
Network Computing Platform
1623-14 Shimotsuruma
Yamato-shi, Kanagawa-ken 242-8502 Japan

jwhaley@alum.mit.edu

Abstract

This paper presents a new combined pointer and escape analysis algorithm for Java programs with unstructured multithreading. The algorithm is based on the abstraction of *parallel interaction graphs*, which characterize the points-to and escape relationships between objects and the ordering relationships between actions performed by multiple parallel threads. To our knowledge, this algorithm is the first interprocedural, flow-sensitive pointer analysis algorithm capable of extracting the points-to relationships generated by the interactions between unstructured parallel threads. It is also the first algorithm capable of analyzing interactions between threads to extract precise escape information even for objects accessible to multiple threads.

We have implemented our analysis in the IBM Jalapeño dynamic compiler for Java and used the analysis results to eliminate redundant synchronization. For our benchmark programs, the thread interaction analysis significantly improves the effectiveness of the synchronization elimination algorithm as compared with previously published techniques, which do not analyze these interactions.

1 Introduction

This paper presents a new, combined pointer and escape analysis algorithm for multithreaded programs. To our knowledge, this algorithm is the first interprocedural, flow-sensitive pointer analysis algorithm for programs with the unstructured form of multithreading present in Java and similar languages. It is also the first algorithm capable of analyzing interactions between threads to extract precise escape information even for objects accessible to multiple threads.

1.1 Analysis Overview

The analysis is based on an abstraction we call *parallel interaction graphs*. The nodes in this graph represent objects; the edges between nodes represent references between objects. For each node, the analysis also records information that characterizes how it escapes the current analysis region. For example, an object escapes if it is reachable from an unanalyzed thread running in parallel with the current thread or returned to an unanalyzed region of the program.

Combining points-to and escape information in the same analysis enables the algorithm to represent all potential interactions between the analyzed and unanalyzed regions of the program. The algorithm represents these interactions, in part, by distinguishing between two kinds of edges: *inside edges*, which represent references created within the currently analyzed region, and *outside edges*, which represent references created outside this region. Each outside edge represents a potential interaction in which the analyzed region reads a reference created in an unanalyzed region. Each inside edge from escaped node represents a potential interaction in which the analyzed region creates a reference that an unanalyzed region may read.

Representing potential interactions with inside and outside edges leads to an analysis that is compositional in two senses:

- **Method Compositionality:** The algorithm analyzes each method once to derive a single parameterized analysis result that records all potential interactions of the method with its callers.[1] At each call site, the algorithm matches outside edges from the callee against inside edges from the caller to compute the effect of the callee on the points-to and escape information of the caller.
- **Thread Compositionality:** The algorithm analyzes each thread once to derive an analysis result that records all of the potential interactions of the thread with other parallel threads. The analysis can then combine analysis results from multiple parallel threads by matching each outside edge from each thread against all corresponding inside edges from parallel threads. The result is a single parallel interaction graph that completely characterizes the points-to and escape information generated by the combined parallel execution of the threads. Unlike previously published algorithms, which use an iterative fixed-point algorithm to compute the interactions [37, 16], the algorithm presented in this paper can compute the interactions between two parallel threads with a single pass over the parallel interaction graphs from the threads.

Finally, the combination of points-to and escape information in the same analysis leads to an algorithm that is

[1] Recursive methods require an iterative algorithm that may analyze methods multiple times to reach a fixed point.

designed to analyze arbitrary regions of complete or incomplete programs, with the analysis result becoming more precise as more of the program is analyzed. At every stage in the analysis, the current parallel interaction graph provides complete information about the points-to relationships for objects that do not escape the currently analyzed region of the program. The algorithm can therefore obtain useful analysis information without analyzing the entire program.

1.2 Analysis Uses

Parallel interaction graphs also record the actions that each thread performs and contain ordering information for these actions relative to the actions performed by other threads. Optimization and analysis algorithms can use this information to determine that actions from different threads can never execute concurrently and are therefore independent. Our compiler uses this ordering information to improve the precision of the thread interaction analysis. It also uses this information to implement a synchronization elimination optimization — if all lock acquire and release actions on a given object are independent, they have no effect on the computation and can be removed.

The analysis also provides information that is generally useful to compilers and program analysis tools for multithreaded programs. Potential applications of our analysis include: sophisticated software engineering tools such as static race detectors and program slicers [28, 36]; memory system optimizations such as prefetching and moving computation to remote data; automatic batching of long latency file system operations; memory bank disambiguation in compilers for distributed memory machines [8]; memory module splitting in compilers that generate hardware directly from high-level languages [6]; lock coarsening [33, 22]; synchronization elimination and stack allocation[41, 10, 12, 15]; and to provide information required to apply traditional compiler optimizations such as constant propagation, common subexpression elimination, register allocation, code motion and induction variable elimination to multithreaded programs.

1.3 Contributions

This paper makes the following contributions:

- **Analysis Algorithm:** It presents a new combined pointer and escape analysis algorithm for multithreaded programs. The algorithm is compositional at both the method and thread levels and is designed to deliver useful information without analyzing the entire program.

- **Analysis Uses:** It shows how to use the action ordering information present in parallel interaction graphs to perform a synchronization elimination optimization.

- **Experimental Results:** It presents experimental results from a prototype implementation of the algorithms. These results show that the algorithm can eliminate a significant number of synchronization operations.

The remainder of the paper is organized as follows. Section 2 presents an example that illustrates how the analysis works. Sections 3 through 11 present the analysis algorithms. Section 14 presents experimental results, Section 15 presents related work, and we conclude in Section 16.

2 Example

In this section we present an example that illustrates how the analysis works. Figure 1 presents the Java code for the example. The `sum` method in the `Sum` class computes the sum of the numbers from `0` to `n`, storing the result into a destination accumulator `a`. It computes this sum by creating a worker thread to compute the sum of the even numbers while it computes the sum of the odd numbers. When they finish, both threads add their contribution into the destination accumulator. The `sum` method first constructs a work vector `v` of `Integers` for the worker thread to sum up, then initializes the worker object to point to the work vector and the destination accumulator. It starts the worker thread running by invoking its `start` method, which invokes the `run` method in a new thread running in parallel with the current thread. This mechanism of initializing a thread object to point to its conceptual parameters is the standard way for Java programs to provide threads with the information they need to initiate their computation.

```
   class Accumulator {
     int value = 0;
     synchronized void add(int v) {
       value += v;
     }
   }
   class Sum {
     public static void sum(int n, Accumulator a) {
1:       Vector v = new Vector();
       for (int i = 0; i < n; i += 2) {
         v.addElement(new Integer(i));
       }
2:       Worker t = new Worker();
       t.init(v,a);
       t.start();
       int s = 0;
       for (int i = 1; i < n; i+= 2) {
         s = s + i;
       }
       a.add(s);
     }
   }
   class Worker extends Thread {
     Vector work;
     Accumulator dest;
     void init(Vector v, Accumulator a) {
       work = v;
       dest = a;
     }
     public void run() {
3:       Enumeration e = work.elements();
       int s = 0;
       while (e.hasMoreElements()) {
         Integer i = (Integer) e.nextElement();
         s = s + i.intValue();
       }
4:       dest.add(s);
     }
   }
```

Figure 1: Sum Example

We contrast the unstructured form of multithreading in this example with the structured, fork-join form of multi-

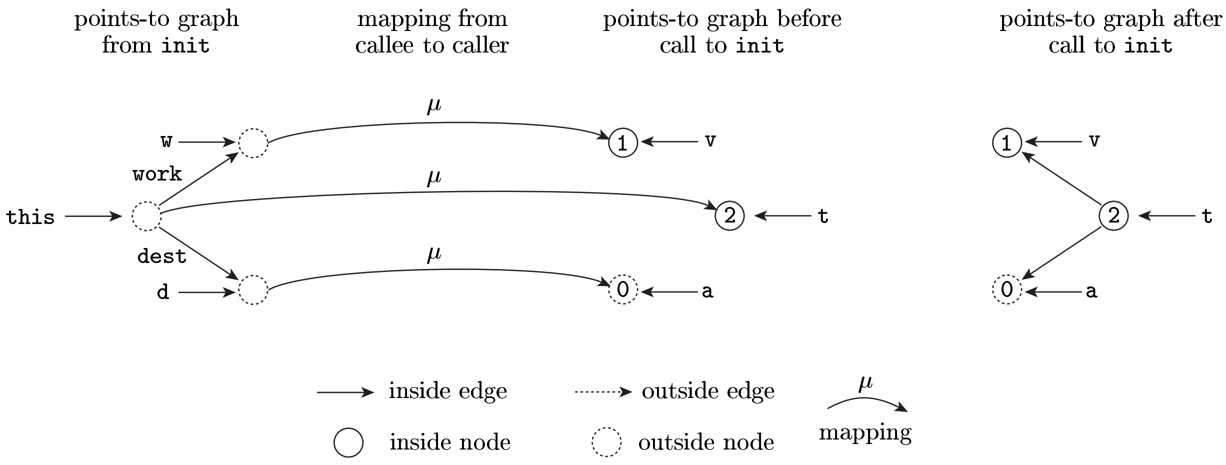


Figure 2: Callee-Caller Interaction Between `init` and `sum`

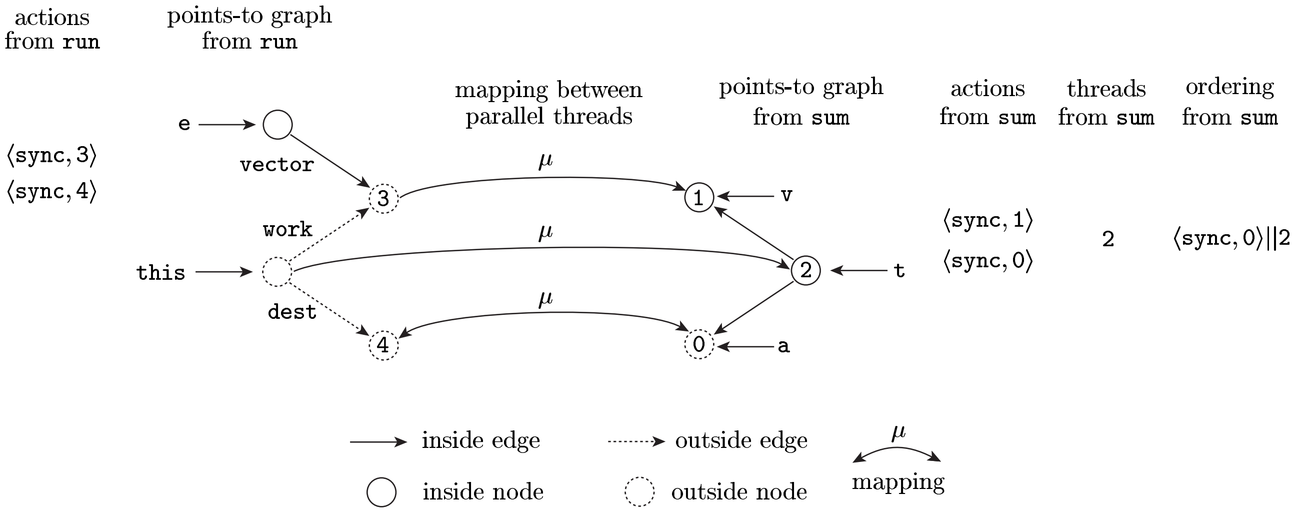


Figure 3: Parallel Thread Interaction Between `run` and `sum`

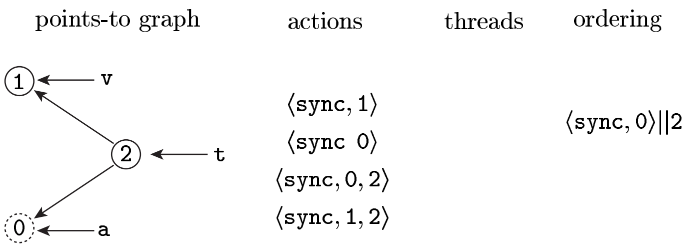


Figure 4: Result of Parallel Thread Interaction

threading found in, for example, the Cilk programming language [11]. Once the thread is the example is created, it executes independently of its parent thread and in parallel with the rest of its parent thread's computation. Cilk threads, on the other hand, must join with their parent thread, completing before their parent thread returns from the procedure in which they were spawned.

We now illustrate the analysis by discussing its operation on this example. We start with the `init` method. This method is passed the work vector and destination accumulator and initializes the worker to point to them. The analysis result for this method is a *parallel interaction graph.* Figure 2 contains the points-to graph from the parallel interaction graph at the end of the `init` method. In general, our points-to graphs contain two kinds of nodes: *inside* nodes, which represent objects created during the computation of the method, and *outside* nodes, which represent objects created outside its computation. All of the nodes in the points-to graph for the `init` method represent the receiver or objects passed into the method as parameters. These nodes are therefore outside nodes. Our points-to graphs also contain two kinds of edges: *inside* edges, which represent references created during the computation of the method, and *outside* edges, which represent references created outside the computation. Because the `init` method reads no references created by other methods or threads, all of the edges in its points-to graph are inside edges.

2.1 Interaction Between Caller and Callee

Figure 2 also presents the points-to graph from the `sum` method just before the call to `init`. This graph contains one outside node (node `0`), which represents the destination accumulator passed as a parameter to `sum`. It also contains two inside nodes — node `1`, which represents the work vector, and node `2`, which represents the worker thread object. Each inside node corresponds to an object creation site and represents all objects created at that site. In the example, we label inside nodes with the line number of the corresponding object creation site in Figure 1.

We next discuss how the analysis combines this points-to graph with the points-to graph from the `init` method to derive the points-to graph after the call to `init`. The algorithm uses the correspondence between the formal and actual parameters to construct a mapping from the outside nodes of the `init` method to the nodes of the `sum` method. This mapping is then used to translate the inside edges from the `init` method into the points-to graph from the `sum` method. Figure 2 presents the result of this mapping, which yields the points-to graph after the call to `init`.

2.2 Interaction Between Parallel Threads

We next discuss how the analysis computes the interaction between the two threads in the example. Figure 3 presents the parallel interaction graph from the end of the worker thread's `run` method. This method loads the `work` and `dest` references from the receiver object. Because these references were created outside the `run` method, the analysis uses an outside edge to represent each reference. Each of these outside edges points to specific kind of outside node called a *load node.* In general, there is one load node for each statement in the program that loads a reference from an escaped object; that load node represents all of the objects to which the reference may point. In Figure 3, we have labeled the load nodes with the number of the corresponding load statement from Figure 1.

In addition to the points-to information, the parallel interaction graph also records the synchronization actions that the method performs and the objects to which the actions are applied. In this case, the `run` method synchronizes on the work vector and the destination accumulator. The actions $\langle \texttt{sync}, 3\rangle$ and $\langle \texttt{sync}, 4\rangle$ record these synchronizations. The synchronization on the work vector happens inside the enumeration's `nextElement` method — Java library classes such as `Vector` often come with the synchronization required for correct execution in the face of concurrent access by parallel threads. In this case, however, the synchronizations are unnecessary. Even though the work vector is accessed by multiple threads, the accesses are separated temporally by thread start events. Among other things, this example will show how the analysis detects the independence of the synchronization actions on the work vector.

We next move to the parallel interaction graph at the end of the `sum` method. In addition to the points-to and action information, the graph records the threads that the method starts and ordering information between the method's action and started threads. In this case, `sum` starts the worker thread, which is represented in the analysis by node `2`. The ordering relation $\langle \texttt{sync}, \texttt{0}\rangle || \texttt{2}$ records the fact that a synchronization action on object `0` (the destination accumulator) may execute in parallel with the actions of the worker thread. Note that there is no parallel ordering relation between the worker thread and `sum`'s synchronization actions on the work vector object. This absence indicates that all of these actions occur before the worker thread starts, and therefore do not execute in parallel with any of the thread's actions.

To model the parallel interaction, the analysis constructs a bidirectional mapping between the nodes of the parallel interaction graphs. Initially, the receiver object of the `run` method is mapped to the worker thread object from the `sum` method. The analysis then matches outside edges from one graph to corresponding inside edges from the other graph, using the match to extend the mapping from outside nodes to inside nodes. When the mapping is complete, it is used to combine the graph from the `run` method into the graph from the `sum` method. Figure 4 presents the combined graph. In addition to the points-to information, the combination algorithm also translates the synchronization actions from the worker thread into the new parallel interaction graph. It tags each action with the thread that performs the action. So, for example, the action $\langle \texttt{sync}, 0, 2\rangle$ indicates that node `2` (the worker thread's node) may perform a synchronization action on an object represented by node `0`.

The `run` method also invokes the `work.elements()` method, which creates an enumerator object to enumerate through elements of the vector. The enumerator object is represented by an inside node. Note that because the enumeration object is captured in the `run` method, it cannot affect the computation outside this method. Therefore, the algorithm does not transfer its inside node into the combined graph.

2.3 Information in the Combined Graph

The compiler can extract the following information from the combined graph. First, the work vector node is captured in this graph. Even though it is accessed by multiple threads, it is not reachable from outside the total computation of the `sum` method. The combined graph therefore completely characterizes the points-to information and actions involving the work vector object. Second, both the thread executing the `sum` method and the worker thread synchronize on the work vector object. But the ordering information indicates that none of these synchronizations can occur concurrently. The synchronization actions have no effect on the computation and can therefore be removed.[2] Finally, both thread synchronize on the destination accumulator object. But in this case, the synchronization actions from the `sum` thread may execute in parallel with the synchronization actions from the worker thread. The compiler will not be able to remove the synchronization from the destination accumulator object.

3 Analysis Abstractions

In this section we formally present the basic abstractions that the analysis uses: the program and object representations, points-to escape graphs, and parallel interaction graphs.

3.1 Program Representation

The algorithm represents the program using the following analysis objects. There is a set $\mathtt{l} \in \mathtt{L}$ of local variables and a set $\mathtt{p} \in \mathtt{P}$ of formal parameter variables. There is one formal parameter variable for each formal parameter of each method in the program. There is also a set $\mathtt{cl} \in \mathtt{CL}$ of class names. The analysis models static class variables using a one-of-a-kind node for each class; the fields of this node are the static class variables for the corresponding class. The analysis therefore treats the class name `cl` as a read-only variable that points to the corresponding one-of-a-kind node that contains the class's static class variables. Together, the local, formal parameter, and class name variables make up the set $\mathtt{v} \in \mathtt{V} = \mathtt{L} \cup \mathtt{P} \cup \mathtt{CL}$ of variables. There is also a set $\mathtt{f} \in \mathtt{F}$ of object fields and a set $\mathtt{op} \in \mathtt{OP}$ of methods. Object fields are accessed using syntax of the form `v.f`. Static class variables are accessed using syntax of the form `cl.f`. Each method has a receiver class `cl` and a formal parameter list $\mathtt{p}_0, \ldots, \mathtt{p}_k$. We adopt the convention that parameter $\mathtt{p}_0$ points to the receiver object of the method.

The algorithm represents the computation of each method using a control flow graph. The nodes of these graphs are statements $\mathtt{st} \in \mathtt{ST}$. We assume the program has been preprocessed so that all statements relevant to the analysis are in one of the following forms:

- A copy statement $\mathtt{l} = \mathtt{v}$.
- A load statement $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$.
- A store statement $\mathtt{l}_1.\mathtt{f} = \mathtt{l}_2$.
- A monitor acquire statement `acquire(l)`.
- A monitor release statement `release(l)`.
- A return statement `return l`, which identifies the return value `l` of the method.
- An object creation site of the form $\mathtt{l} = \mathtt{new}\ \mathtt{cl}$.
- A method invocation site $m \in M$ of the form $\mathtt{l} = \mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$.
- A thread start site of the form `l.start()`.

[2] To satisfy the Java memory model, the compiler may have to leave memory barriers behind at the old synchronization points.

The analysis represents the control flow relationships between statements as follows: pred(`st`) is the set of statements that may execute immediately before `st`, and succ(`st`) is the set of statements that may execute immediately after `st`. There are two program points for each statement `st`, the program point $\bullet\mathtt{st}$ immediately before `st` executes, and the program point $\mathtt{st}\bullet$ immediately after `st` executes. The control flow graph for each method `op` starts with an enter statement $\mathtt{enter}_{\mathtt{op}}$ and ends with an exit statement $\mathtt{exit}_{\mathtt{op}}$.

The interprocedural analysis uses call graph information to compute sets of methods that may be invoked at method invocation sites. For each method invocation site $m \in M$, callees(m) is the set of methods that m may invoke. Given a method `op`, callers(`op`) is the set of method invocation sites that may invoke `op`. The current implementation obtains this call graph information using a variant of class hierarchy analysis [17], but the algorithm can use any conservative approximation to the actual call graph generated when the program runs.

3.2 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes. There are several kinds of nodes:

- There is a set N_I of inside nodes. Inside nodes represent *inside objects*, which are objects created within the current analysis scope and accessed via references created within the current analysis scope. This set consists of two subsets:
  - Nodes in $\underline{N_I}$ represent objects created by the current thread. There is one node in $\underline{N_I}$ for each object creation site; that node represents all objects created at that site by the current thread.
  - Nodes in $\overline{N_I}$ represent objects created by threads running in parallel with the current thread. There is one node in $\overline{N_I}$ for each object creation site; that node represents all objects created at that site by threads running in parallel with the current thread.

  Two nodes $n_1 \in \underline{N_I}$ and $n_2 \in \overline{N_I}$ are corresponding nodes if they represent objects created at the same object creation site.

  In Java, each thread corresponds to an object that implements the `Runnable` interface. The set $N_T \subseteq N_I$ of runnable nodes represents runnable objects. $\underline{N_T} \subseteq \underline{N_I}$ represents runnable objects created by the current thread, and $\overline{N_T} \subseteq \overline{N_I}$ represents runnable objects created by threads running in parallel with the current thread. $N_T = \underline{N_T} \cup \overline{N_T}$.

- There is a set N_O of outside nodes. Outside nodes represent *outside objects*, which are objects created outside the current analysis scope or accessed via references created outside the current analysis scope. This set consists of several subsets:
  - There is a set N_L of load nodes. When a load statement executes, it loads a value from a field in an object. If the loaded value is a reference, the analysis must represent the object that the reference points to. Each load node represents outside objects whose references are loaded by the corresponding load statement. There are two kinds of load nodes:

* $\underline{N_L}$ contains one node for each load statement in the program. That node represents outside objects whose references are loaded at that statement by the current thread.
* $\overline{N_L}$ contains one node for each load statement in the program. That node represents outside objects whose references are loaded at that statement by threads running in parallel with the current thread.

Two nodes $n_1 \in \underline{N_L}$ and $n_2 \in \overline{N_L}$ are corresponding nodes if they represent outside objects whose references are loaded at the same load statement.

– There is a set of return nodes N_R. When the algorithm skips the analysis of a method invocation site, it uses a return node to represent the return value of the method invoked at that site. There are two kinds of return nodes:

* $\underline{N_R}$ contains one node for each skipped method invocation site in the program. That node represents objects returned by methods invoked at that site by the current thread.
* $\overline{N_R}$ contains one node for each skipped method invocation site in the program. That node represents objects returned by methods invoked at that site by threads running in parallel with the current thread.

Two nodes $n_1 \in \underline{N_R}$ and $n_2 \in \overline{N_R}$ are corresponding nodes if they represent objects returned at the same skipped method invocation site.

– N_P: There is one parameter node $n \in N_P$ for each formal parameter in the program. Each parameter node represents the object that its parameter points to during the execution of the analyzed method. The receiver object is treated as the first parameter of each method. Given a parameter `p`, the corresponding parameter node is $n_{\texttt{p}}$. There is always an inside edge from `p` to $n_{\texttt{p}}$.

– N_C: There is one class node $n \in N_C$ for each class in the program. The fields of this node represent the static class variables of its class. Given a class `cl`, the corresponding class node is $n_{\texttt{cl}}$. There is always an inside edge from `cl` to $n_{\texttt{cl}}$.

The set $\underline{N} = \underline{N_I} \cup \underline{N_L} \cup \underline{N_R}$, and the set $\overline{N} = \overline{N_I} \cup \overline{N_L} \cup \overline{N_R}$. Given a node $n \in \underline{N}$, $\overline{n}$ represents the corresponding node in $\overline{N}$. Given a node $n \in \overline{N}$, $\underline{n}$ represents the corresponding node in $\underline{N}$.

The analysis represents each array with a single node. This node has a field `elements`, which represents all of the elements of the array. Because the points-to information for all of the array elements is merged into this field, the analysis does not make a distinction between different elements of the same array.

3.3 Points-To Escape Graphs

A points-to escape graph is a quadruple of the form $\langle O, I, e, r \rangle$, where

- $O \subseteq (N \times \texttt{F}) \times N_L$ is a set of outside edges. Outside edges represent references created outside the current analysis scope, either by the caller, by a thread running in parallel with the current thread, or by an unanalyzed invoked method.
- $I \subseteq ((N \times \texttt{F}) \times N) \cup (\texttt{V} \times N)$ is a set of inside edges. Inside edges represent references created inside the current analysis scope.
- $e : N \to 2^{N_P \cup N_C \cup N_T \cup M}$ is an escape function that records the escape information for each node. A node escapes if it is reachable from a parameter node $n_P \in N_P$, a static class variable represented by a field of a class node $n_C \in N_C$, a thread node $n_T \in N_T$ running in parallel with the current thread, or an object passed as a parameter to or returned from an unanalyzed method invocation site.
- $r \subseteq N$ is a return set that represents the set of objects that may be returned by the currently analyzed method. All nodes in the return set escape to the caller of the analyzed method.

Both O and I are graphs with edges labeled with a field from F. We define the following operations on nodes of the graphs:

$$\begin{array}{rcl} \text{edgesTo}(I, n) &=& \{\langle \texttt{v}, n\rangle \in I\} \cup \{\langle\langle n', \texttt{f}\rangle, n\rangle \in I\} \\ \text{edgesFrom}(I, \texttt{v}) &=& \{\langle \texttt{v}, n\rangle \in I\} \\ \text{edgesFrom}(I, n) &=& \{\langle\langle n, \texttt{f}\rangle, n'\rangle \in I\} \\ \text{edges}(I, \texttt{v}) &=& \text{edgesFrom}(I, \texttt{v}) \\ \text{edges}(I, n) &=& \text{edgesTo}(I, n) \cup \text{edgesFrom}(I, n) \\ I(\texttt{v}) &=& \{n . \langle \texttt{v}, n\rangle \in I\} \\ I(n, \texttt{f}) &=& \{n' . \langle\langle n, \texttt{f}\rangle, n'\rangle \in I\} \end{array}$$

For each node n, the escape function $e(n)$ and the return set r together record all of the different ways the node (and the objects that it represents) may escape from the current analysis scope. Here are the possibilities:

- If a parameter node $n_{\texttt{p}} \in e(n)$, then n represents an object that may be reachable from `p`.
- If a class node $n_{\texttt{cl}} \in e(n)$, then n represents an object that may be reachable from one of the static class variables of the class `cl`.
- If a thread node $n_T \in e(n)$, then n represents an object that may be reachable from a runnable object represented by n_T.
- If a method invocation site $m \in e(n)$, then n represents an object that may be reachable from the parameters or the return value of an unanalyzed method invoked at m.
- If $n \in r$, then n represents an object that may be returned to the caller of the analyzed method.

The escape information must satisfy the escape information propagation invariant that if n_1 points to n_2, then n_2 escapes in at least all of the ways that n_1 escapes. We formalize this invariant with the following inference rule, which states that if there is an edge from n_1 to n_2, $e(n_1) \subseteq e(n_2)$. When the analysis adds an edge to the points-to escape graph, it may need to update the escape information.

$$\frac{\langle\langle n_1, \texttt{f}\rangle, n_2\rangle \in O \cup I}{e(n_1) \subseteq e(n_2)}$$

We say that a node n_1 violates the propagation constraint if there is an edge from n_1 to n_2 and $e(n_1) \not\subseteq e(n_2)$. During the analysis of a method, the algorithm may add edges to the points-to escape graph. These edges may make the updated nodes (the nodes that the new edges point from) temporarily violate the propagation constraint. Whenever it adds a new edge, the analysis uses the propagate($\langle O, I, e, r\rangle, S$) algorithm in Figure 5 to propagate escape information from the nodes in S to restore the invariant and produce a new escape function e' that satisfies the propagation constraint. This algorithm takes a points-to escape graph and a set S of nodes that may violate the constraint, then uses a worklist approach to propagate the escape information from the updated nodes to the other nodes in the graph.

```
propagate(⟨O, I, e, r⟩, S)
  Initialize worklist and new escape function
  e' = e
  W = S
  while (W ≠ ∅) do
    Remove a node from worklist
    W = W − {n1}
    Propagate escape information to all nodes
      that n1 points to
    for all ⟨⟨n1, f⟩, n2⟩ ∈ O ∪ I do
      Restore constraint for n2
      e'(n2) = e'(n2) ∪ e'(n1)
      if e'(n2) changed then
        W = W ∪ {n2}
  return(e')
```

Figure 5: Escape Information Propagation Algorithm

Given our abstraction of points-to escape graphs, we can define the concepts of escaped and captured nodes as follows:

- escaped($\langle O, I, e, r\rangle, n$) if $e(n) \neq \emptyset$ or $n \in r$, and
- captured($\langle O, I, e, r\rangle, n$) if $e(n) = \emptyset$ and $n \notin r$.

4 Actions

The algorithm is designed to record various actions that the program performs on objects. Each action consists of an action label that identifies the kind of action performed, a node that represents the object on which the action was performed, and an optional thread node that represents the thread that performed the action. For the purposes of this paper, the set of action labels is $\mathtt{b} \in \mathtt{B} = \{\mathtt{ld}, \mathtt{sync}\}$. The set of actions is $a \in A = (\mathtt{B} \times N) \cup (\mathtt{B} \times N \times N_T)$. Here is the meaning of the actions:

- $\langle \mathtt{sync}, n\rangle$ records a synchronization action (either a monitor acquire or release) performed by the current thread on an object represented by n.
- $\langle \mathtt{sync}, n, n_T\rangle$ records a synchronization action performed by a thread represented by n_T.
- $\langle \mathtt{ld}, n\rangle$ records a load by the current thread on an escaped node. The result of the load is a reference to an object represented by the load node n.
- $\langle \mathtt{ld}, n, n_T\rangle$ records a load performed by the thread n_T.

It is straightforward to augment the set of action labels and the analysis to record arbitrary actions such as reading and writing objects or invoking a given method on an object. It is also straightforward to generalize the set of actions to include actions performed on multiple objects.

5 Parallel Interaction Graphs

The algorithm uses a dataflow analysis to generate, at each program point in the method, a parallel interaction graph $\langle G, \tau, \alpha, \pi\rangle$.

- G is a points-to escape graph that summarizes the points-to and escape information for the current thread.
- The parallel thread map $\tau : N_T \to \{0, 1, 2\}$ counts the number of instances of each thread that may execute in parallel with the current thread at the current program point. If $\tau(n) = 1$, then at most one instance of n may execute in parallel with the current thread at the current program point; if $\tau(n) = 2$, then multiple instances of n may execute in parallel with the current thread at the current program point. We define the following operations:
  - $x \oplus y = \min(2, x + y)$
  - $x \ominus y = \begin{cases} 2 & \text{if } x \geq 2 \\ \max(0, x - y) & \text{otherwise} \end{cases}$
- The action set $\alpha \subseteq A$ records the set of actions executed by the analyzed computation.
- The parallel action relation $\pi \subseteq A \times N_T$ records ordering information between the actions of the current thread and threads that execute in parallel with the current thread. Specifically, $\langle a, n_T\rangle \in \pi$ if a may have happened after at least one of the thread objects represented by n_T started executing. In this case, the actions of a thread object represented by n_T may affect a.

remove($\langle G, \tau, \alpha, \pi\rangle, S$) denotes the parallel interaction graph obtained by removing all of the nodes in S from $\langle G, \tau, \alpha, \pi\rangle$.

The analysis uses parallel interaction graphs to compute the interactions between parallel threads. During the analysis, one of the threads is the current thread; conceptually, nodes from the other threads move into the context of the current thread. In the analyis context of the current thread, all of the nodes from the other threads come from outside the current thread. The analysis models this by replacing each node from an other thread with its corresponding version from outside the current thread. Given a parallel interaction graph $\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle$, $\langle\langle \overline{O}, \overline{I}, \overline{e}, \overline{r}\rangle, \overline{\tau}, \overline{\alpha}, \overline{\pi}\rangle$ is the parallel interaction graph with nodes replaced with the corresponding nodes from outside the current thread, defined as follows:

$$
\begin{aligned}
\mu(n) &= \begin{cases} \overline{n} & \text{if } n \in \underline{N} \\ n & \text{otherwise} \end{cases} \\
\sigma(S) &= \{\mu(n).n \in S\} \\
\overline{O} &= \{\langle\langle \mu(n_1), \mathtt{f}\rangle, \mu(n_2)\rangle.\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O\} \\
\overline{I} &= \{\langle\langle \mu(n_1), \mathtt{f}\rangle, \mu(n_2)\rangle.\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I\} \cup \\
&\quad\ \{\langle \mathtt{v}, \mu(n)\rangle.\langle \mathtt{v}, n\rangle \in I\} \\
\overline{e}(n) &= \begin{cases} \sigma(e(n)) \cup \sigma(e(\underline{n})) & \text{if } n \in \overline{N} \\ \emptyset & \text{if } n \in \underline{N} \\ \sigma(e(n)) & \text{otherwise} \end{cases} \\
\overline{r} &= \{\mu(n).n \in r\} \\
\overline{\tau}(n) &= \begin{cases} \tau(n) \oplus \tau(\underline{n}) & \text{if } n \in \overline{N} \\ 0 & \text{otherwise} \end{cases} \\
\mu_A(a) &= \begin{cases} \langle \mathtt{b}, \mu(n)\rangle & \text{if } a = \langle \mathtt{b}, n\rangle \\ \langle \mathtt{b}, \mu(n), \mu(n_T)\rangle & \text{if } a = \langle \mathtt{b}, n, n_T\rangle \end{cases} \\
\overline{\alpha} &= \{\mu_A(a).a \in \alpha\} \\
\overline{\pi} &= \{\langle \mu_A(a), \mu(n)\rangle.\langle a, n\rangle \in \pi\}
\end{aligned}
$$

The following operation removes a set of nodes S from a parallel interaction graph $\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle$.

$$\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle = \text{remove}(\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle, S)$$

where

$$\begin{aligned}
S' &= (N - S) \\
O' &= O \cap ((S' \times F) \times S') \\
I' &= I \cap ((S' \times F) \times S') \\
e'(n) &= e(n) \cap (S' \cup M) \\
r' &= r \cap S' \\
\tau'(n) &= \begin{cases} \tau(n) & \text{if } n \in S' \\ 0 & \text{otherwise} \end{cases} \\
\alpha' &= \alpha \cap ((\mathtt{B} \times S') \cup (\mathtt{B} \times S' \times S')) \\
\pi' &= \pi \cap (((\mathtt{B} \times S') \cup (\mathtt{B} \times S' \times S')) \times S')
\end{aligned}$$

6 Intraprocedural Analysis

The analysis of each method `op` starts with the initial parallel interaction graph $\langle\langle O_0, I_0, e_0, r_0\rangle, \tau_0, \alpha_0, \pi_0\rangle$, defined as follows:

- In I_0, each formal parameter points to its corresponding parameter node and each class points to its corresponding class node.

$$I_0 = \{\langle \mathtt{p}, n_{\mathtt{p}}\rangle . \mathtt{p} \in \mathtt{P}\} \cup \{\langle \mathtt{cl}, n_{\mathtt{cl}}\rangle . \mathtt{cl} \in \mathtt{CL}\}$$

- The initial set of outside edges is empty: $O_0 = \emptyset$
- The initial escape function e_0 is set up so that each parameter or class node is marked as escaping via itself.

$$e_0(n) = \begin{cases} \{\mathtt{n}\} & \text{if } n \in N_P \cup N_C \\ \emptyset & \text{otherwise} \end{cases}$$

- The initial return set and action set are empty: $r_0 = \emptyset$, and $\alpha_0 = \emptyset$.
- Initially there are no threads running in parallel with the current thread. $\forall n_T \in N_T . \tau_0(n_T) = 0$.
- The initial parallel action relation is empty: $\pi_0 = \emptyset$.

The algorithm analyzes the method under the assumption that the parameters and static class variables all point to different objects. If the method may be invoked in a calling context in which some of these pointers point to the same object, this object will be represented by multiple nodes during the analysis of the method. In this case, the analysis described below in Section 9 will merge the corresponding outside objects when it combines the final analysis result for the method into the calling context at the method invocation site. Because the combination algorithm retains all of the edges from the merged objects, it conservatively models the actual effect of the method.

The intraprocedural analysis is a dataflow analysis that propagates parallel interaction graphs through the statements of the method's control flow graph. The transfer function $\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle = [\![\mathtt{st}]\!](\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle)$ defines the effect of each statement `st` on the current parallel interaction graph. Most of the statements first kill a set of inside edges, then generate additional inside and outside edges. Figure 6 graphically presents the rules that determine the sets of generated edges for the different kinds of statements. Each row in this figure contains four items: a statement, a graphical representation of existing edges, a graphical representation of the existing edges plus the new edges that the statement generates, and 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. We would like to point out several aspects of the intraprocedural analysis:

- **`start` Statements:** At each `start` statement of the form `l.start()`, `l` may point to several thread nodes. The analysis adds all of these nodes to the new parallel thread map.
- **Synchronization:** The transfer function for synchronization statements adds a synchronization action to the new parallel interaction graph. This synchronization action is recorded as executing in parallel with all of the thread nodes in the current parallel thread map. This ordering information is used later in the analysis to help determine if synchronization actions on a given node are independent.
- **Outside Edges:** A load statement may add an outside edge to the current parallel interaction graph. In this case, the transfer function also records the fact that the outside edge is created in parallel with all of the thread nodes in the parallel thread map. This ordering information is used during the thread interaction algorithm to ensure that these outside edges are not matched with inside edges from threads whose execution starts after the execution of the load statement that generated the outside edge.

We next present the dataflow analysis framework from the intraprocedural analysis. This framework includes the transfer functions for the basic statements and the definition of the confluence operator at merge points in the control-flow graph.

6.1 Copy Statements

A copy statement of the form `l = v` makes `l` point to the object that `v` points to. The transfer function updates I to reflect this change by killing the current set of edges from `l`, then generating additional inside edges from `l` to all of the nodes that `v` points to.

$$\begin{aligned}
\text{Kill}_I &= \text{edges}(I, \mathtt{l}) \\
\text{Gen}_I &= \{\mathtt{l}\} \times I(\mathtt{v}) \\
I' &= (I - \text{Kill}_I) \cup \text{Gen}_I
\end{aligned}$$

6.2 Load Statements

A load statement of the form $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$ makes $\mathtt{l}_1$ point to the object that $\mathtt{l}_2.\mathtt{f}$ points to. The analysis models this change by constructing a set S of nodes that represent all of the objects to which $\mathtt{l}_2.\mathtt{f}$ may point, then generating additional inside edges from $\mathtt{l}_1$ to every node in this set.

All nodes accessible via inside edges from $\mathtt{l}_2.\mathtt{f}$ should clearly be in S. But if $\mathtt{l}_2$ points to an escaped node, other parts of the program such as the caller or threads executing in parallel with the current thread can access the referenced object and store values in its fields. In particular, the value in $\mathtt{l}_2.\mathtt{f}$ may have been written by the caller or a thread running in parallel with the current thread — in other words, $\mathtt{l}_2.\mathtt{f}$ may contain a reference created outside of the current analysis scope. The analysis uses an outside edge to model

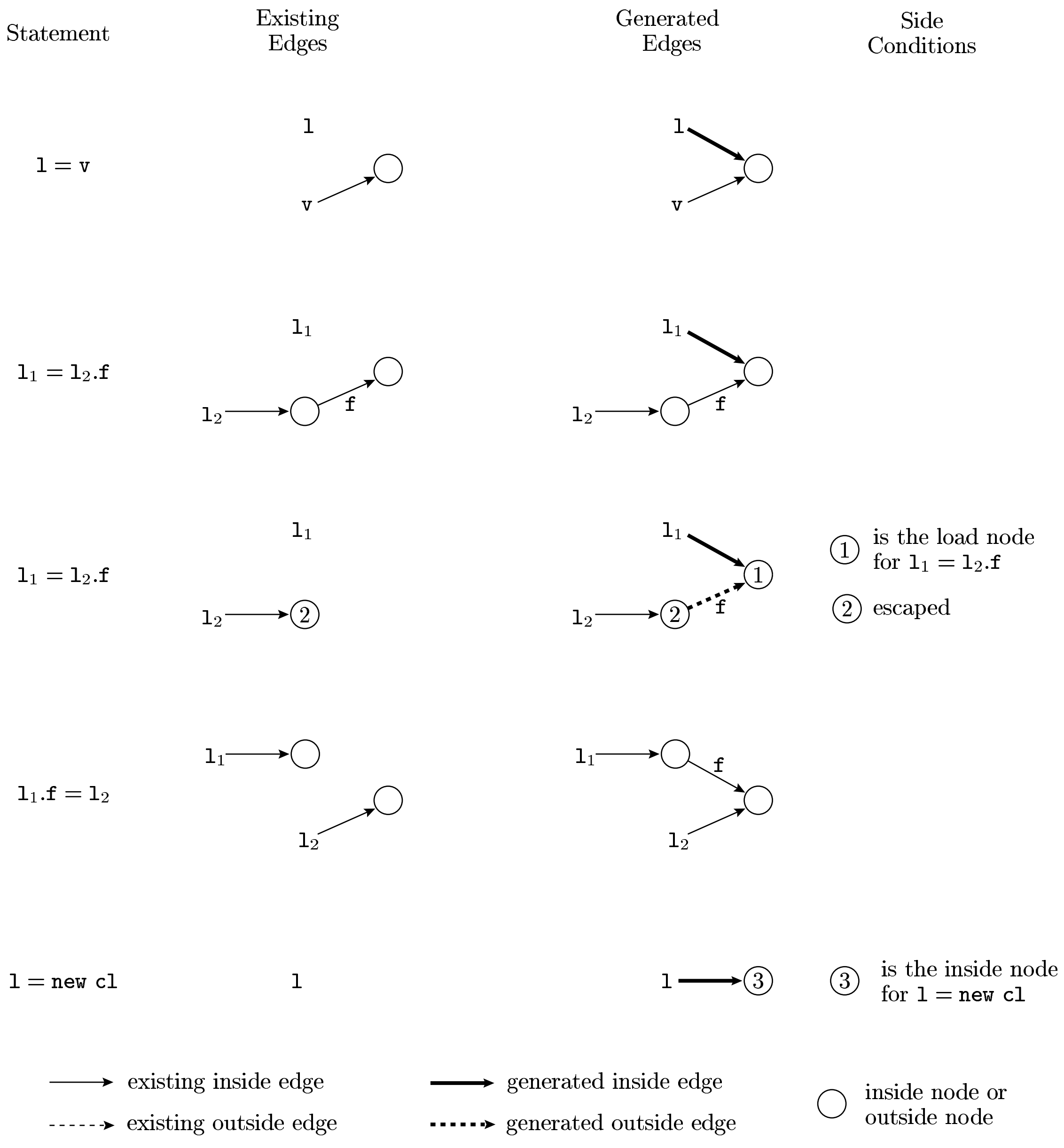


Figure 6: Generated Edges for Basic Statements

this reference. The outside edge points to the load node for the load statement, which is the outside node that represents the objects that the reference may point to.

The analysis must therefore consider two cases: the case when $\mathtt{l}_2$ does not point to an escaped node, and the case when $\mathtt{l}_2$ does point to an escaped node. The algorithm determines which case applies by computing S_E, the set of escaped nodes to which $\mathtt{l}_2$ points. S_I is the set of nodes accessible via inside edges from $\mathtt{l}_2.\mathtt{f}$.

$$\begin{array}{rcl} S_E & = & \{n_2 \in I(\mathtt{l}_2).\mathrm{escaped}(\langle O, I, e, r\rangle, n_2)\} \\ S_I & = & \cup\{I(n_2, \mathtt{f}).n_2 \in I(\mathtt{l}_2)\} \end{array}$$

If $S_E = \emptyset$ (i.e., $\mathtt{l}_2$ does not point to an escaped node), $S = S_I$ and the transfer function simply kills all edges from $\mathtt{l}_1$, then generates inside edges from $\mathtt{l}_1$ to all of the nodes in S.

$$\begin{array}{rcl} \mathrm{Kill}_I & = & \mathrm{edges}(I, \mathtt{l}_1) \\ \mathrm{Gen}_I & = & \{\mathtt{l}_1\} \times S \\ I' & = & (I - \mathrm{Kill}_I) \cup \mathrm{Gen}_I \end{array}$$

If $S_E \neq \emptyset$ (i.e., $\mathtt{l}_2$ points to at least one escaped node), $S = S_I \cup \{n_L\}$, where n_L is the load node for the load statement. In addition to killing all edges from $\mathtt{l}_1$, then generating inside edges from $\mathtt{l}_1$ to all of the nodes in S, the transfer function also generates outside edges from the escaped nodes to n_L and propagates the escape information from the escaped nodes through n_L. It also generates a load action $\langle \mathtt{ld}, n\rangle$ and updates the parallel action relation to record the fact that the action may execute in parallel with thread objects represented by the current set of parallel thread nodes. In this case, the new outside edges may represent references created by these parallel thread objects.

$$\begin{array}{rcl} \mathrm{Kill}_I & = & \mathrm{edges}(I, \mathtt{l}_1) \\ \mathrm{Gen}_I & = & \{\mathtt{l}_1\} \times S \\ I' & = & (I - \mathrm{Kill}_I) \cup \mathrm{Gen}_I \\ \mathrm{Gen}_O & = & (S_E \times \{\mathtt{f}\}) \times \{n_L\} \\ O' & = & O \cup \mathrm{Gen}_O \\ e' & = & \mathrm{propagate}(\langle O', I', e, r\rangle, S_E) \\ \alpha' & = & \alpha \cup \{\langle \mathtt{ld}, n_L\rangle\} \\ \pi' & = & \pi \cup (\{\langle \mathtt{ld}, n_L\rangle\} \times \{n_T.\tau(n_T) > 0\}) \end{array}$$

6.3 Store Statements

A store statement of the form $\mathtt{l}_1.\mathtt{f} = \mathtt{l}_2$ finds the object to which $\mathtt{l}_1$ points, then makes the $\mathtt{f}$ field of this object point to same object as $\mathtt{l}_2$. The analysis models the effect of this assignment by finding the set of nodes that $\mathtt{l}_1$ points to, then generating inside edges from all of these nodes to the nodes that $\mathtt{l}_2$ points to. It also propagates the escape information from all of the nodes that $\mathtt{l}_1$ points to through the nodes that $\mathtt{l}_2$ points to.

$$\begin{array}{rcl} \mathrm{Gen}_I & = & (I(\mathtt{l}_1) \times \{\mathtt{f}\}) \times I(\mathtt{l}_2) \\ I' & = & I \cup \mathrm{Gen}_I \\ e' & = & \mathrm{propagate}(\langle O', I', e, r\rangle, I(\mathtt{l}_1)) \end{array}$$

6.4 Acquire and Release Statements

An acquire statement of the form $\mathtt{acquire}(\mathtt{l})$ finds the object to which $\mathtt{l}$ points, then acquires that object's lock. A release statement of the form $\mathtt{release}(\mathtt{l})$ finds the object to which $\mathtt{l}$ points, then releases that object's lock. The analysis models the effect of these statements by finding the set of nodes that $\mathtt{l}$ points to, then recording synchronization actions on all of these nodes.

$$\begin{array}{rcl} \alpha' & = & \alpha \cup (\{\mathtt{sync}\} \times I(\mathtt{l})) \\ \pi' & = & \pi \cup ((\{\mathtt{sync}\} \times I(\mathtt{l})) \times \{n_T.\tau(n_T) > 0\}) \end{array}$$

6.5 Object Creation Sites

An object creation site of the form $\mathtt{l} = \mathtt{new\ cl}$ allocates a new object and makes $\mathtt{l}$ point to the object. The analysis represents all objects allocated at a specific creation site with the creation site's inside node n. The transfer function models the effect of the statement by killing all edges from $\mathtt{l}$, then generating an inside edge from $\mathtt{l}$ to n.

$$\begin{array}{rcl} \mathrm{Kill}_I & = & \mathrm{edges}(I, \mathtt{l}) \\ \mathrm{Gen}_I & = & \{\langle \mathtt{l}, n\rangle\} \\ I' & = & (I - \mathrm{Kill}_I) \cup \mathrm{Gen}_I \end{array}$$

6.6 Return Statements

A return statement $\mathtt{return\ l}$ specifies the return value for the method. The immediate successor of each return statement is the exit statement of the method. The analysis models the effect of the return statement by updating r to include all of the nodes that $\mathtt{l}$ points to.

$$r' = I(\mathtt{l})$$

6.7 Control-Flow Join Points

To analyze a statement, the algorithm first computes the join of the parallel interaction graphs flowing into the statement from all of its predecessors. It then applies the transfer function to obtain a new parallel interaction graph at the point after the statement. The join operation $\sqcup$ is defined as follows.

$$\begin{array}{l} \langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle = \\ \quad \langle\langle O_1, I_1, e_1, r_1\rangle, \tau_1, \alpha_1, \pi_1\rangle \sqcup \langle\langle O_2, I_2, e_2, r_2\rangle, \tau_2, \alpha_2, \pi_2\rangle \end{array}$$

where $O = O_1 \cup O_2$, $I = I_1 \cup I_2$, $\forall n \in N.e(n) = e_1(n) \cup e_2(n)$, $r = r_1 \cup r_2$, $\forall n \in N.\tau(n) = \max(\tau_1(n), \tau_2(n))$, $\alpha = \alpha_1 \cup \alpha_2$, and $\pi = \pi_1 \cup \pi_2$.

The corresponding partial order $\sqsubseteq$ is

$$\langle\langle O_1, I_1, e_1, r_1\rangle, \tau_1, \alpha_1, \pi_1\rangle \sqsubseteq \langle\langle O_2, I_2, e_2, r_2\rangle, \tau_2, \alpha_2, \pi_2\rangle$$

if $O_1 \subseteq O_2$, $I_1 \subseteq I_2$, $\forall n \in N.e_1(n) \subseteq e_2(n)$, $r_1 \subseteq r_2$, $\forall n \in N.\tau_1(n) \leq \tau_2(n)$, $\alpha_1 \subseteq \alpha_2$, and $\pi_1 \subseteq \pi_2$. Bottom is $\langle\langle\emptyset, \emptyset, e_\perp, \emptyset\rangle, \tau_\perp, \emptyset, \emptyset\rangle$, where $\forall n \in N.e_\perp(n) = \emptyset$ and $\forall n \in N.\tau_\perp(n) = 0$.

6.8 Analysis Results

The analysis of each method produces analysis results $\beta(\bullet\mathtt{st})$ and $\beta(\mathtt{st}\bullet)$ before and after each statement $\mathtt{st}$ in the method's control flow graph. The analysis result β satisfies the following equations:

$$\begin{array}{rcl} \beta(\bullet\mathtt{enter}_{\mathtt{op}}) & = & \langle\langle O_0, I_0, e_0, r_0\rangle, \tau_0, \alpha_0, \pi_0\rangle \\ \beta(\bullet\mathtt{st}) & = & \sqcup\{\beta(\mathtt{st}'\bullet).\mathtt{st}' \in \mathrm{pred}(\mathtt{st})\} \\ \beta(\mathtt{st}\bullet) & = & [\![\mathtt{st}]\!](\beta(\bullet\mathtt{st})) \end{array}$$

The final analysis result of method $\mathtt{op}$ is the analysis result at the program point after the exit node, i.e., $\beta(\mathtt{exit}_{\mathtt{op}}\bullet)$. As described below in Section 9.4, the analysis solves these equations using a standard worklist algorithm.

7 Matching Inside and Outside Edges

In the interprocedural analysis, outside edges in the callee's parallel interaction graph represent inside edges in the caller's parallel interaction graph. To compute the effect of a method call, the analysis matches the callee's outside edges against the corresponding inside edges from the caller. In the inter-thread analysis, outside edges in the parallel interaction graphs of each thread represent inside edges in the parallel interaction graph of the other thread. To compute the interactions, the analysis matches outside edges from each thread against inside edges from the other thread. The matching process is conceptually similar in both cases. This section discusses the matching algorithm we use for both the interprocedural and the inter-thread analyses.

The matching algorithm takes two points-to escape graphs $\langle O_i, I_i, e_i, r_i \rangle$ $(i \in \{1,2\})$ and two initial mappings $\mu_i : N \to N$. It produces two new mappings $\mu_i' : N \to N$ that extend the initial mappings to map the outside nodes in each graph to the corresponding nodes in the other graph.

$$\langle \mu_1', \mu_2' \rangle = \mathrm{match}(\langle O_1, I_1, e_1, r_1 \rangle, \langle O_2, I_2, e_2, r_2 \rangle, \mu_1, \mu_2)$$

We formulate the mapping using set inclusion constraints [1]. This formulation enables us to present a compact, simple specification of the mapping result using a set of constraint rules. Figure 7 presents the constraints that the matching algorithm must satisfy. Note that these contraints use the notation $\bar{i}$ to represent the complement of i; i.e. $\overline{1} = 2$, $\overline{2} = 1$. The constraints basically specify that if an outside edge from one graph matches an inside edge from the other graph, then the mappings must map the outside edge's node to the inside edge's node. This node mapping potentially enables more edge matchings.

Figure 9 presents the algorithm that solves these constraints. It operates by repeatedly finding a node in one graph that is already mapped to a node in the other graph. It then checks if there is an inside edge from one of these nodes that it can match up with a corresponding outside edge from the other node. If so, it updates one of the mappings to reflect the fact that the node that the outside edge points to is mapped to the node that the inside edge points to.

match($\langle O_1, I_1, e_1, r_1 \rangle, \langle O_2, I_2, e_2, r_2 \rangle, \mu_1, \mu_2$)
    Initialize worklists and results
    for $i = 1, 2$ do
        $\mu_i' = \mu_i$
        $W_i = \{\langle n_1, n_3 \rangle . n_3 \in \mu_i(n_1)\}$
        $D_i = \emptyset$
    while choose $\langle n_1, n_3 \rangle \in W_i$ do
        Remove a pair from worklist
        $W_i = W_i - \{\langle n_1, n_3 \rangle\}$
        $D_i = D_i \cup \{\langle n_1, n_3 \rangle\}$
        Check outside edges for rule 2
        for all $\langle \langle n_1, \mathtt{f} \rangle, n_2 \rangle \in O_i$ do
            for all $\langle \langle n_3, \mathtt{f} \rangle, n_4 \rangle \in I_{\bar{i}}$ do
                $\mu_i'(n_2) = \mu_i'(n_2) \cup \{n_4\}$
                if $\langle n_2, n_4 \rangle \notin D_i$ then
                    $W_i = W_i \cup \{\langle n_2, n_4 \rangle\}$
        Check inside edges for rule 3
        for all $\langle \langle n_1, \mathtt{f} \rangle, n_2 \rangle \in I_i$ do
            for all $\langle \langle n_3, \mathtt{f} \rangle, n_4 \rangle \in O_{\bar{i}}$ do
                $\mu_{\bar{i}}'(n_4) = \mu_{\bar{i}}'(n_4) \cup \{n_2\}$
                if $\langle n_4, n_2 \rangle \notin D_{\bar{i}}$ then
                    $W_{\bar{i}} = W_{\bar{i}} \cup \langle n_4, n_2 \rangle$
    return $\langle \mu_1', \mu_2' \rangle$

Figure 9: Algorithm for Matching Inside and Outside Nodes

8 Combining Points-To Escape Graphs

Once the matching algorithm has set up the mapping between inside and outside nodes in the two graphs, the combination algorithm uses the mapping to generate a new points-to escape graph that reflects the final points-to and escape relationships generated by the interaction.

The combination algorithm takes two points-to escape graphs $\langle O_i, I_i, e_i, r_i \rangle$ $(i \in \{1,2\})$ and two initial mappings $\mu_i : N \to N$. It produces the final points-to escape graph $\langle O', I', e', r' \rangle$. The combination algorithm performs two basic tasks: it traces out reachable edges and nodes from the two input points-to escape graphs so that they are present in the final graph, and it uses the mappings between outside and inside nodes to translate inside edges into the final graph.

$$\begin{aligned}&\langle \langle O', I', e', r' \rangle, \mu_1', \mu_2' \rangle = \\ &\quad \mathrm{combine}(\langle O_1, I_1, e_1, r_1 \rangle, \langle O_2, I_2, e_2, r_2 \rangle, \mu_1, \mu_2)\end{aligned}$$

As for the mapping algorithm, we specify the combination result using a system of set inclusion constraints. Figure 8 presents the constraints that specify the result of the combination algorithm. These constraints extend the initial mappings to two new mappings $\mu_i' : N \to N$. These new mappings have the property that $n \in \mu_i'(n)$ if n is reachable in the final points-to escape graph and should be present in that graph. These constraints start with a set of nodes that will be mapped into the combined graph. They then trace out the reachable nodes to determine the complete set of nodes that should be present in the combined graph. Figures 10 and 11 present an algorithm for solving the constraint system in Figure 8. We highlight several properties of this algorithm:

- **Base Nodes:** The analysis starts with a set of base nodes mapped into the combined graph. In the case of caller/callee interaction, all of the nodes from the caller are mapped directly into the combined graph. The class nodes from the callee are also mapped directly, while the parameter nodes are mapped indirectly to model the semantics of method invocation.
- **Inside Nodes:** An inside node from one of the parallel interaction graphs is present in the combined graph if it is reachable from the base nodes.
- **Inside Edges:** If two nodes are mapped into the combined graph, the analysis uses the mapping to translate insided edges between the two nodes into the graph.
- **Load Nodes:** A load node is mapped into the combined graph if it is reachable from an escaped base node.
- **Outside Edges:** Outside edges are translated into the combined graph if the node that they come from is mapped into the graph and at least one of the nodes that it maps to is escaped in the graph.

8.0.1 Constraint Solution Algorithm

We next discuss the constraint solution algorithm in Figures 10 and 11 for the constraint system in Figure 8. The algorithm directly reflects the structure of the inference rules. At each step it detects an inference rule antecedent that becomes true, then takes action to ensure that the consequent is also true.

Constraint		Existing Edges and Mapping	New Mapping

$$\mu_i(n) \subseteq \mu_i'(n) \qquad (1)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_i, \langle\langle n_3, \mathtt{f}\rangle, n_4\rangle \in I_{\bar{i}}, n_3 \in \mu_i'(n_1)}{n_4 \in \mu_i'(n_2)} \qquad (2)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_i, \langle\langle n_3, \mathtt{f}\rangle, n_4\rangle \in O_{\bar{i}}, n_3 \in \mu_i'(n_1)}{n_2 \in \mu_{\bar{i}}'(n_4)} \qquad (3)$$

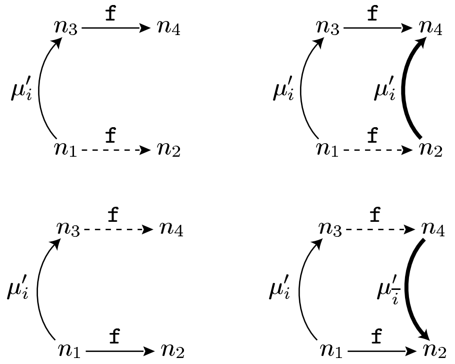


Figure 7: Constraints for Matching Inside and Outside Edges

Constraint		Existing Edges and Mapping	New Edges and Mapping	Side Conditions

$$\mu_i(n) \subseteq \mu_i'(n) \qquad (4)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_i, n \in \mu_i'(n_1), n_2 \in N_I \cup N_R}{n_2 \in \mu_i'(n_2)} \qquad (5)$$

Side condition: $n_2 \in N_I \cup N_R$

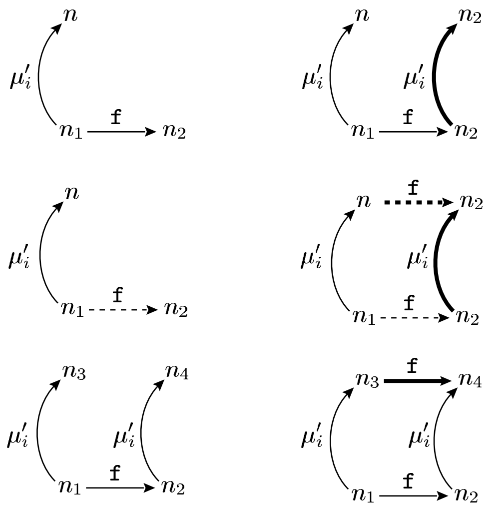
$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_i, n \in \mu_i'(n_1), \quad \text{escaped}(\langle O', I', e', \emptyset\rangle, n)}{n_2 \in \mu_i'(n_2), \langle\langle n, \mathtt{f}\rangle, n_2\rangle \in O'} \qquad (6)$$

Side condition: n escaped

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_i}{(\mu_i'(n_1) \times \{\mathtt{f}\}) \times \mu_i'(n_2) \subseteq I'} \qquad (7)$$

$$\frac{n \in e_i(n_1), n_2 \in \mu_i'(n_1)}{n \in e'(n_2)} \qquad (8)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O' \cup I'}{e'(n_1) \subseteq e'(n_2)} \qquad (9)$$



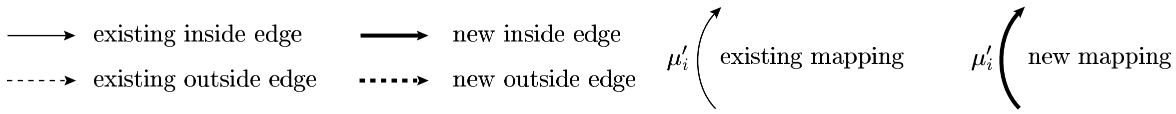


Figure 8: Constraints for Combining Points-to Escape Graphs

The mapNode(n_1, n, i) procedure in Figure 10 is invoked whenever the algorithm maps a node n_1 from points-to escape graph i to a node n in the new graph. It first matches inside edges involving n_1 in points-to escape graph i to inside edges involving n in the new graph. The procedure checks any edges to n_1 that have already been previously translated into the new graph to see if they should also be translated to point to n in the new graph. It also checks all of the inside edges from n_1 to see if they should be translated into the new graph. The procedure then checks outside edges from n_1 to see if they should be translated into the new graph. Finally, the procedure updates the new escape function e' to reflect the effect of the newly mapped nodes and newly translated edges.

Figure 11 presents the driver for the constraint solution algorithm. It maintains a worklist W_i^I of inside nodes from graph i that should be mapped into the new graph, and a worklist W_i^O of outside edges that should be translated into the new graph if the edge's source node is escaped in the new graph. When the algorithm processes a node from W_i^I, it calls mapNode to map that node into the new graph. When the algorithm processes an outside edge from W_i^O, it translates the edge into the new graph and maps the edge's target node into the new graph.

There is a slight complication in the algorithm. As the algorithm executes, it periodically translates inside edges into the new graph. Whenever a node n_1 from graph i is mapped to n, the algorithm translates each inside edge $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ from graph i into the new graph. This translation process inserts a corresponding inside edge from n to each node that n_2 maps to; i.e., to each node in $\mu_i'(n_2)$. The algorithm must ensure that when it completes, there is one such edge for each node in the final set of nodes $\mu_i'(n_2)$. But when the algorithm first translates $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ into the new graph, $\mu_i'(n_2)$ may not be complete. In this case, the algorithm will eventually map n_2 to more nodes, increasing the set of nodes in $\mu_i'(n_2)$. There should be edges from n to all of the nodes in the final $\mu_i'(n_2)$, not just to those node that were present in $\mu_i'(n_2)$ when the algorithm mapped $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ into the new graph.

The algorithm ensures that all of these edges are present in the final graph by building a set $\delta_i(n_2)$ of delayed actions. Each delayed action consists of a node in the new graph and a field in that node. Whenever the node n_2 is mapped to a new node n' (i.e., the algorithm sets $\mu_i'(n_2) = \mu_i'(n_2) \cup \{n'\}$), the algorithm establishes a new inside edge for each delayed action. The new edge goes from the node in the action to the newly mapped node n'. These edges ensure that the final set of inside edges satisfies the constraints.

9 Interprocedural Analysis

The interprocedural analysis algorithm propagates parallel interaction graphs from callees to callers. At thread start sites, the analysis adds the nodes that represent the started thread to the parallel thread map. It also marks the started thread nodes as escaping via themselves and propagates the escape information. At each method invocation site, the analysis has the option of either skipping the site or analyzing the site. If it skips the site, it marks all of the parameters and the return value as permanently escaping down into the site.

mapNode(n_1, n, i)
  if $\langle n_1, n, i\rangle \notin D$ then
    $D = D \cup \{\langle n_1, n, i\rangle\}$
    $\mu_i'(n_1) = \mu_i'(n_1) \cup \{n\}$
    Add delayed inside edges for rule 7
    $I' = I' \cup \delta_i(n_1) \times \{n\}$
    $S = \{n' \in N . \langle n', \mathtt{f}\rangle \in \delta(n_1)\}$
    Check inside edges for rules 5 and 7
    for all $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_i$
      Add inside edges for rule 7
      $I' = I' \cup \{\langle n, \mathtt{f}\rangle\} \times \mu_i'(n_2)$
      $S = S \cup \{n\}$
      $\delta_i(n_2) = \delta_i(n_2) \cup \{\langle n, \mathtt{f}\rangle\}$
      Check conditions for rule 5
      if $n_2 \in N_I \cup N_R$
        $W_i^I = W_i^I \cup \{n_2\}$
    Check outside edges for rule 6
    for all $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_i$
      $W_i^O = W_i^O \cup \{\langle\langle n, \mathtt{f}\rangle, n_2\rangle\}$
    Update escape information for rules 8 and 9
    $e'(n) = e'(n) \cup e_i(n_1)$
    $e' = \text{propagate}(\langle O', I', e', r'\rangle, S)$

Figure 10: Algorithm for Mapping One Node to Another

combine$(\langle O_1, I_1, e_1, r_1\rangle, \langle O_2, I_2, e_2, r_2\rangle, \mu_1, \mu_2)$
  Initialize worklists and results
  $D = \emptyset$
  for $i = 1, 2$ do
    $\langle W_i^I, W_i^O\rangle = \langle\emptyset, \emptyset\rangle$
    for all $n \in N$ do $\delta_i(n) = \emptyset$
    for all $n \in N$ do $\mu_i'(n) = \emptyset$
  $\langle O', I', r'\rangle = \langle\emptyset, \emptyset, \emptyset\rangle$
  for all $n \in N$ do $e'(n) = \emptyset$
  Call mapNode *for existing mappings*
  for all $\langle n_1, n, i\rangle$ such that $n \in \mu_i(n_1)$ do
    mapNode(n_1, n, i)
  done = false
  do
    if choose $n \in W_i^I$ then
      $W_i^I = W_i^I - \{n\}$
      mapNode(n, n, i)
    else if choose $\langle\langle n, \mathtt{f}\rangle, n_1\rangle \in W_i^O$ such that
        escaped$(\langle O', I', e', \emptyset\rangle, n)$ then
      $W_i^I = W_i^I - \{\langle\langle n, \mathtt{f}\rangle, n_1\rangle\}$
      Update outside edges for rule 6
      $O' = O' \cup \{\langle\langle n, \mathtt{f}\rangle, n_1\rangle\}$
      mapNode(n_1, n_1, i)
    else done = true
  while not done
  return $\langle\langle O', I', e', r'\rangle, \mu_1', \mu_2'\rangle$

Figure 11: Algorithm for Combining Points-to Escape Graphs

9.1 Thread Start Sites

To simplify the presentation of the analysis, we assume that at each thread start site `l.start()`, $I(l) \subseteq N_T$. This is almost invariably the case in practice — most threads are started in the same method in which they are allocated. The algorithm adds the thread nodes that `l` points to into the set of parallel threads. It also makes the escape function for each node $n_T \in I(\mathtt{l})$ include n_T, and propagates the new escape information.

$$\begin{aligned}
\tau'(n_T) &= \begin{cases} \tau(n_T) \oplus 1 & \text{if } n_T \in I(\mathtt{l}) \\ \tau(n_T) & \text{otherwise} \end{cases} \\
e_T(n) &= \begin{cases} e(n) \cup \{n\} & \text{if } n \in I(\mathtt{l}) \\ e(n) & \text{otherwise} \end{cases} \\
e' &= \text{propagate}(\langle O, I, e_T, r\rangle, I(\mathtt{l}))
\end{aligned}$$

9.2 Skipped Method Invocation Sites

The transfer function for a skipped method invocation site is defined as follows. Given a skipped method invocation site m of the form $\mathtt{l} = \mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$ with return node n_R and a current parallel interaction graph $\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle$, the parallel interaction graph $\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle = [\![m]\!](\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle)$ after the site is defined as follows:

$$\begin{aligned}
I' &= (I - \text{edges}(I, \mathtt{l})) \cup \{\langle \mathtt{l}, n_R\rangle\} \\
O' &= O \\
e' &= \text{propagate}(\langle O', I', e_m, r\rangle, S_m) \\
r' &= r
\end{aligned}$$

where $S_m = \cup\{I(\mathtt{l}_i).0 \leq i \leq k\}$ and

$$e_m(n) = \begin{cases} e(n) \cup \{m\} & \text{if } n \in S_m \text{ or } n = n_R \\ e(n) & \text{otherwise} \end{cases}$$

Recall that the return node n_R is an outside node used to represent the return value of the invoked method.

9.3 Analyzed Method Invocation Sites

Given an analyzed method invocation site m and a current parallel interaction graph $\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle$, the new parallel interaction graph $\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle = [\![m]\!](\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle)$ after the site is defined as follows:

$$\begin{aligned}
&\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle = \\
&\quad \sqcup \{\text{mapUp}(\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle, m, \mathtt{op}).\mathtt{op} \in \text{callees}(m)\}
\end{aligned}$$

Figure 12 presents the combination algorithm, which performs the following steps:

- It retrieves the analysis result from the exit statement of the invoked method `op`.
- It builds an initial mapping. This mapping maps the parameter nodes from the callee to the corresponding nodes in the caller that represent the actual parameters, and the class nodes to themselves.
- It uses the initial mapping to match outside edges in the callee to the corresponding inside edges in the caller. The result is a mapping from the outside nodes of the callee to the corresponding nodes in the caller.
- It uses the result mapping to combine the caller and callee graphs, generating the new parallel interaction graph. In addition to combining the points-to and escape information, the analysis must also translate the actions and parallel threads from the callee into the caller. It must also record the fact that all of the actions from the callee occur in parallel with all of the parallel threads from the caller.

The transfer function itself merges the combined results from all potentially invoked methods to derive the points-to escape graph at the point after the method invocation site.

mapUp($\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle, m, \mathtt{op}$)

Assume m of the form $\mathtt{l} = \mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$

Assume `op` has formal parameters $\mathtt{p}_0, \ldots, \mathtt{p}_k$

Extract the analysis result from the invoked method

$\langle\langle O_R, I_R, e_R, r_R\rangle, \tau_R, \alpha_R, \pi_R\rangle = \beta(\mathtt{exit}_{\mathtt{op}}\bullet)$

Compute initial mappings

$$\mu_C(n) = \begin{cases} \{n\} & \text{if } n \in N_C \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mu^m_{\mathtt{op}}(n) = \begin{cases} \{n\} & \text{if } n \in N_C \\ I(l_i) & \text{if } n = n_{\mathtt{p}_i} \\ \emptyset & \text{otherwise} \end{cases}$$

Match outside nodes from callee to caller nodes

$\langle \mu_1, \mu_2\rangle = \text{match}(\langle \emptyset, I, e, r\rangle, \langle O_R, I_R, e_R, r_R\rangle, \mu_C, \mu^m_{\mathtt{op}})$

Compute mappings for combination

$$\mu(n) = \begin{cases} \mu_1(n) \cup \{n\} & \text{if } \tau_R(n) > 0 \text{ or } n \in r \text{ or} \\ & \quad \exists \mathtt{v} \in \mathtt{V}.\langle \mathtt{v}, n\rangle \in I \\ \mu_1(n) & \text{otherwise} \end{cases}$$

$$\mu_R(n) = \begin{cases} \mu_2(n) \cup \{n\} & \text{if } \tau_R(n) > 0 \text{ or} \\ & \quad n \in r_R - (N_P \cup N_L) \\ \mu_2(n) & \text{otherwise} \end{cases}$$

$e'_R(n) = e_R(n) - \mathtt{P}$

Combine points-to escape graphs

$\langle\langle O', I_C, e', r_C\rangle, \mu'_1, \mu'_2\rangle =$
combine($\langle O, I, e, r\rangle, \langle O_R, I_R, e'_R, r_R\rangle, \mu, \mu_R$)

Generate the final set of inside edges and return set

$I' = I_C \cup (I - \text{edges}(\mathtt{l})) \cup (\{l\} \times \bigcup_{n \in r_R} \mu'_2(n))$

$r' = r$

Compute new thread map

$\tau'(n) = \tau(n) \oplus \tau_R(n)$

Compute action mapping from callee to caller

$$\mu_A(a) = \begin{cases} \{\mathtt{b}\} \times \mu'_2(n) & \text{if } a = \langle \mathtt{b}, n\rangle \\ \{\mathtt{b}\} \times \mu'_2(n) \times \{n_T\} & \text{if } a = \langle \mathtt{b}, n, n_T\rangle \end{cases}$$

Combine actions from caller and callee

$\alpha' = \alpha \cup (\bigcup_{a \in \alpha_R} \mu_A(a))$

Compute new parallel action relation

$$\begin{aligned}\pi' = \pi \cup (&\bigcup_{\langle a, n_T\rangle \in \pi_R} \mu_A(a) \times \{n_T\}) \cup \\ (&\bigcup_{a \in \alpha_R} \mu_A(a) \times \{n_T.\tau(n_T) > 0\})\end{aligned}$$

Return the new parallel interaction graph

return $\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle$

Figure 12: Callee/Caller Interaction Algorithm

9.4 Fixed-Point Analysis Algorithm

Figure 13 presents the interprocedural fixed-point algorithm that the compiler uses to generate the interprocedural analysis results. It uses a worklist of pending statements to solve the combined intraprocedural and interprocedural dataflow equations. At each step, it removes a statement from the worklist and updates the analysis results before and after the statement. If the analysis result after the statement changed, it inserts all of its successors (or for exit nodes, all of the callers of its method) into the worklist. As specified, the algorithm is therefore both intraprocedural and interprocedural.

Initialize analysis results
for all $\texttt{st} \in \texttt{ST}$ do
  $\beta(\bullet\texttt{st}) = \beta(\texttt{st}\bullet) = \langle\langle\emptyset, \emptyset, e_\emptyset, \emptyset\rangle, \tau_0, \emptyset, \emptyset\rangle$
    where $e_\emptyset(n) = \emptyset$ for all $n \in N$ and
      $\tau_0(n_T) = 0$ for all $n_T \in N_T$
for all $\texttt{op} \in \texttt{OP}$ do
  $\beta(\bullet\texttt{enter}_{\texttt{op}}) = \langle\langle O_0, I_0, e_0, r_0\rangle, \tau_0, \alpha_0, \pi_0\rangle$
Initialize the worklist
$W = \{\texttt{enter}_{\texttt{op}}.\texttt{op} \in \texttt{OP}\}$
while $(W \neq \emptyset)$ do
  Remove a statement from worklist
  $W = W - \{\texttt{st}\}$
  Process the statement
  $\beta(\bullet\texttt{st}) = \sqcup\{\beta(\texttt{st}'\bullet).\texttt{st}' \in \text{pred}(\texttt{st})\}$
  $\beta(\texttt{st}\bullet) = [\![\texttt{st}]\!](\alpha(\beta\texttt{st}))$
  if $\beta(\texttt{st}\bullet)$ changed then
    Put potentially affected statements onto worklist
    $W = W \cup \text{succ}(\texttt{st})$
    if $\texttt{st} \equiv \texttt{exit}_{\texttt{op}}$ then
      $W = W \cup \text{callers}(\texttt{op})$

Figure 13: Fixed-Point Analysis Algorithm

The order in which the algorithm analyzes methods can have a significant impact on the analysis. For non-recursive methods, a bottom-up analysis of the program yields the full result with one analysis per method. For recursive methods, the analysis results must be iteratively recomputed within each strongly connected component of the call graph using the current best result until the analysis reaches a fixed point.

It is possible to extend the algorithm so that it initially skips the analysis of method invocation sites. If the analysis result is not precise enough, it can incrementally increase the precision by analyzing method invocation sites that it originally skipped. The algorithm will then propagate the new, more precise result to update the analysis results at affected program points.

10 Inter-thread Analysis

The interprocedural algorithm described above generates a parallel interaction graph for every point in the program. This graph records all of the points-to relationships created by the current thread and all of the potential interactions of that thread with other threads. The thread interaction algorithm resolves the potential interactions by computing which interactions may actually occur between the current thread and the parallel threads that it (transitively) starts. The basic idea is to repeatedly match corresponding inside and outside edges from parallel threads. The result is a single parallel interaction graph that summarizes the combined effect of the parallel threads on the points-to and escape information at that point.

10.1 Thread Interaction

We next discuss the interaction algorithm for parallel threads. The interaction takes place between a starter thread (a thread that starts a parallel thread) and a startee thread (the thread that is started). The interaction algorithm is given the parallel interaction graph $\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle$ from the starter thread, a node n_T that represents the startee thread, and a run method $\texttt{op}$ with receiver object $\texttt{p}_0$ that runs when the thread object represented by n_T starts.

$$\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle = \text{interact}(\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle, n_T, \texttt{op})$$

Figure 14 presents the interaction algorithm. The algorithm performs the following steps:

- It extracts the final analysis result for the startee thread. This is the analysis result after the exit node of the startee thread's `run` method.
- It matches corresponding inside and outside edges from the two threads. All of the outside edges from the startee thread participate in the matching. Outside edges from the starter participate only if they represent loads that may have occurred after the startee thread began its execution. The algorithm uses the event ordering information to determine which of these outside edges should participate.
- It combines the two parallel interaction graphs to generate the final parallel interaction graph. In addition to combining the points-to and escape information, the analysis must also translate the actions and started threads from the two graphs into the combined graphs. It must also update the ordering information as follows:
  - Each thread has ordering information that specifies which of its actions may execute in parallel with the threads that it starts. For both threads, this ordering information is translated into the new points-to escape graph.
  - All actions from the starter thread that occur in parallel with the startee thread also occur in parallel with all of the startee's threads.
  - All actions from the startee thread occur in parallel with all of the starter's threads.

Note that because the parallel interaction graphs represent all potential interactions between threads, the algorithm can compute the actual interactions with a single pass over the two graphs.

10.2 Resolution

To generate the final parallel interaction graph that summarizes all of the interactions, the algorithm resolves all of the interactions between the parallel threads. The resolution algorithm repeatedly takes the current parallel interaction graph, chooses one of the startee threads, and computes the interactions between the current graph and the startee thread to derive a new current graph. It continues this process until it reaches a fixed point. In the absence of loop or recursively generated concurrency, the algorithm reaches a fixed point after processing each thread once. The result interact$(\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle, n_T, \texttt{op})$

Assume $\texttt{p}_0$ represents receiver of $\texttt{op}$

Extract the analysis result for the parallel thread

$\langle\langle O_R, I_R, e_R, r_R\rangle, \tau_R, \alpha_R, \pi_R\rangle = \upsilon(\texttt{exit}_{\texttt{op}}\bullet)$

Compute outside edges from starter thread that participate in mapping

$$O_{n_T} = \{\langle\langle n_1, \texttt{f}\rangle, n_2\rangle . \langle\langle \texttt{ld}, n_2\rangle, n_T\rangle \in \pi \text{ or } \langle\langle \texttt{ld}, n_2, n\rangle, n_T\rangle \in \pi\}$$

Compute initial mappings for the match

$$\mu_C(n) = \begin{cases} \{n\} & \text{if } n \in N_C \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mu_{\texttt{op}}^{n_T}(n) = \begin{cases} \{n\} & \text{if } n \in N_C \\ \{n_T\} & \text{if } n = n_{\texttt{p}_0} \\ \emptyset & \text{otherwise} \end{cases}$$

Match corresponding inside and outside edges

$$\langle \mu_1, \mu_2\rangle = \text{match}(\langle O_{n_T}, I, e, r\rangle, \langle \overline{O_R}, \overline{I_R}, \overline{e_R}, \overline{r_R}\rangle, \mu_C, \mu_{\texttt{op}}^{n_T})$$

Compute mappings for the combination

$$\mu(n) = \begin{cases} \mu_1(n) \cup \{n\} & \text{if } \tau(n) > 0 \text{ or } n \in r \text{ or } \exists \texttt{v} \in \texttt{V}.\langle \texttt{v}, n\rangle \in I \\ \mu_1(n) & \text{otherwise} \end{cases}$$

$$\mu_R(n) = \begin{cases} \mu_2(n) \cup \{n\} & \text{if } \overline{\tau_R}(n) > 0 \\ \mu_2(n) & \text{otherwise} \end{cases}$$

$$\overline{e_R}'(n) = \overline{e_R}(n) - \texttt{P}$$

Combine the two parallel interaction graphs

$$\langle\langle O', I_C, e', r_C\rangle, \mu_1', \mu_2'\rangle = \text{combine}(\langle O, I, e, r\rangle, \langle \overline{O_R}, \overline{I_R}, \overline{e_R}', \overline{r_R}\rangle, \mu, \mu_R)$$

$$I' = I_C \cup \bigcup_{\langle \texttt{v}, n\rangle \in I} \{\texttt{v}\} \times \mu_1'(n)$$

$$r' = \bigcup_{n \in r} \mu_1'(n)$$

Compute action mappings for combination

$$\mu_1^A(a) = \begin{cases} \{\texttt{b}\} \times \mu_1'(n) & \text{if } a = \langle \texttt{b}, n\rangle \\ \{\texttt{b}\} \times \mu_1'(n) \times \{n_T'\} & \text{if } a = \langle \texttt{b}, n, n_T'\rangle \end{cases}$$

$$\mu_2^A(a) = \begin{cases} \{\texttt{b}\} \times \mu_2'(n) & \text{if } a = \langle \texttt{b}, n\rangle \\ \{\texttt{b}\} \times \mu_2'(n) \times \{n_T'\} & \text{if } a = \langle \texttt{b}, n, n_T'\rangle \end{cases}$$

Compute new parallel thread map

$$\tau_{n_T}(n) = \begin{cases} \tau(n) \ominus 1 & \text{if } n = n_T \\ \tau(n) \oplus \overline{\tau_R}(n) & \text{otherwise} \end{cases}$$

$$\tau'(n) = \tau_{n_T}(n) \oplus \overline{\tau_R}(n)$$

Compute combined action sets

$$\alpha' = (\bigcup_{a \in \alpha} \mu_1^A(a)) \cup (\bigcup_{\langle \texttt{b}, n\rangle \in \overline{\alpha_R}} \{\texttt{b}\} \times \mu_2^A(n) \times \{n_T\}) \cup (\bigcup_{\langle \texttt{b}, n, n_T'\rangle \in \overline{\alpha_R}} \{\texttt{b}\} \times \mu_2^A(n) \times \{n_T'\})$$

Compute new parallel action relation

$$\pi' = (\bigcup_{\langle a, n_T'\rangle \in \pi} \mu_1^A(a) \times \{n_T'\}) \cup (\bigcup_{\langle a, n_T\rangle \in \pi} \mu_1^A(a) \times \{n_T' . \overline{\tau_R}(n_T') > 0\}) \cup (\bigcup_{\langle a, n_T'\rangle \in \overline{\pi_R}} \mu_2^A(a) \times \{n_T'\}) \cup (\bigcup_{a \in \overline{\alpha_R}} \mu_2^A(a) \times \{n_T' . \tau_{n_T}(n_T') > 0\})$$

Return the new parallel interaction graph

return $\langle\langle O', I', e', r'\rangle, \tau', \alpha', \pi'\rangle$

Figure 14: Parallel Thread Interaction Algorithm

of the resolution algorithm resolve$(\langle G, \tau, \alpha, \pi\rangle)$ must satisfy the following equation:

$$\text{resolve}(\langle G, \tau, \alpha, \pi\rangle) = \bigsqcup_{\langle n_T, \texttt{op}\rangle \in S} \{\text{resolve}(\text{interact}(\langle G, \tau_{n_T}, \alpha, \pi\rangle, n_T, \texttt{op}))\}$$

where

- $S = \{\langle n_T, \texttt{op}\rangle . \tau(n_T) > 0 \text{ and } \texttt{op} \in \text{run}(n_T)\}$
- $\tau_{n_T}(n) = \begin{cases} \tau(n) \ominus 1 & \text{if } n = n_T \\ \tau(n) & \text{otherwise} \end{cases}$

To perform the resolution, the analysis requires information about the correspondence between thread nodes and the `run` method that executes when the `start` method is invoked on an object that the node represents. Given a thread node $n \in N_T$ (a node that represents runnable objects), run(n) is the set of `run` methods that may execute when the `start` method is invoked on an object represented by n. In the current compiler, run(n) is computed using the declared type of n. Figure 15 presents a fixed-point algorithm that computes resolve$(\langle G, \tau, \alpha, \pi\rangle)$.

resolve$(\langle G, \tau, \alpha, \pi\rangle)$

$\quad \langle G', \tau', \alpha', \pi'\rangle = \langle G, \tau, \alpha, \pi\rangle$

$\quad S = \emptyset$

$\quad$ while there exists $n_T \in N_T$ such that

$\quad\quad \tau'(n_T) > 0$ and $\langle n_T, \tau', \alpha', \pi'\rangle \notin S$ do

$\quad\quad S = S \cup \{\langle n_T, \tau', \alpha', \pi'\rangle\}$

$\quad\quad$ choose n_T such that $\tau'(n_T) > 0$ and $\langle n_T, \tau', \alpha', \pi'\rangle \notin S$

$\quad\quad \langle G', \tau', \alpha', \pi'\rangle = \bigsqcup_{\texttt{op} \in \text{run}(n_T)} \text{interact}(\langle G', \tau'_{n_T}, \alpha', \pi'\rangle, n_T, \texttt{op})$

$\quad\quad$ if G' changed then $S = \emptyset$

$\quad$ return$(\langle G', \tau', \alpha', \pi'\rangle)$

Figure 15: Fixed-Point Resolution Algorithm

Given a statement `st`, it is possible to compute a single parallel interaction graph $\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle$ that completely summarizes the points-to and escape relationships after `st` as follows:

$$\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle = \text{trim}(\text{resolve}(\beta(\texttt{st}\bullet)))$$

where

- $S = \{n \in N_L . e(n) - N_T = \emptyset\}$
- trim$(\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle) =$ remove$(\langle\langle O, I, e', r\rangle, \tau, \alpha, \pi\rangle, S)$
- $e'(n) = e(n) - N_T$.

The algorithm trims off any outside edges that come from nodes that escape only because they are reachable from thread nodes. It also removes all thread nodes from the escape function. The resolved graph already contains all of the possible interactions that may affect nodes that escape only via other threads.

For the program point at the exit of the `main` method, the analysis may also trim off outside edges that come from nodes that escape only because they are reachable from static class variables. The resolved graph at this program point contains all of the possible interactions that may affect these nodes. In this graph, the only source of uncertaintly comes from nodes passed into or returned from unanalyzed method invocation sites.

$$\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle = \text{trimMain}(\text{resolve}(\beta(\texttt{exit}_{\texttt{main}}\bullet)))$$

where

- $S = \{n \in N_L . e(n) - (N_T \cup N_C) = \emptyset\}$
- $\text{trim}(\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle) = \text{remove}(\langle\langle O, I, e', r\rangle, \tau, \alpha, \pi\rangle, S)$
- $e'(n) = e(n) - (N_T \cup N_C)$.

11 Independence Testing

The independence testing algorithm finds objects that are captured at the end of a method, using either the resolved graph (as discussed in Section 10.2) or the single thread interprocedural analysis result (as discussed in Section 9). If an object is captured in either graph, it is unreachable outside the method. In this case, the graph completely summarizes all of the actions that threads may perform on the object. The algorithm uses the parallel action relation to determine if two conflicting actions may occur concurrently. If not, the actions are independent and the compiler can eliminate all synchronization on the objects that the node represents. Given a parallel interaction graph $\langle G, \tau, \alpha, \pi\rangle$ from the `exit` node of a method and a captured node n, the algorithm in Figure 16 tests if all of the synchronization actions on n are independent.

independent($\langle G, \tau, \alpha, \pi\rangle, n$)
  Check if the node is escaped in G
  if escaped(G, n) then return false
  Find all `sync` *actions on n*
  $S = \{\langle \texttt{sync}, n\rangle \in \alpha\} \cup \{\langle \texttt{sync}, n, n_T\rangle \in \alpha\}$
  Check if one of the actions a in S may execute in parallel with a thread n_T that also synchronizes on n
  if $\exists a \in S, n_T \in N_T . \langle a, n_T\rangle \in \pi$ and $\langle \texttt{sync}, n, n_T\rangle \in S$ then
    return false
  else return true

Figure 16: Independence Testing Algorithm

The compiler tests all inside nodes for independence in the analysis result $\beta(\texttt{exit}_{\texttt{op}}\bullet)$ for all methods `op`. It also tests all inside nodes in the resolved analysis result $\text{trim}(\text{resolve}(\beta(\texttt{exit}_{\texttt{op}}\bullet)))$ for methods `op` that contain thread creation sites and in the resolved analysis result $\text{trimMain}(\text{resolve}(\beta(\texttt{exit}_{\texttt{main}}\bullet)))$ at the end of the `main` method.

12 Resolving Outside Nodes

It is possible to augment the algorithm so that it records, for each outside node, all of nodes that it represents during the analysis. This information allows the algorithm to go back to the analysis results generated at the various program points and resolve each outside node to the set of inside nodes that it represents during the analysis. The basic idea is to generate a set of inclusion constraint systems. There is one system Ω_m for each method invocation site m and one system $\Omega_{\texttt{op}}$ for each method `op`. These systems specify, for each node n, a map set $\omega(n)$ of nodes that n is mapped to during the analysis of the corresponding method invocation site or method. These systems are specified using set inclusion constraints of the forms $\omega(n_1) \subseteq \omega(n_2)$, which specifies that the map set for n_2 includes the map set for n_1. We use the notation $\omega_m(n)$ to indicate the solution of the constraint system Ω_m for n, and similarly $\omega_{\texttt{op}}(n)$ for the solution of $\Omega_{\texttt{op}}$ for n. The initial map set Ω_0 consists of the set of constraints $\{n\} \subseteq \omega(n)$, which specifies that each node is in its map set. The constraint systems can be solved by a simple constraint propagation algorithm [24].

We define the constraint systems using the mappings generated by the algorithms in Figures 12 and 14. Specifically, $\mu^m_{\texttt{op}}(n) = \mu'_2(n)$, where μ'_2 is the mapping computed by the algorithm in Figure 12 for the method `op` invoked at method invocation site m, and $\mu_{\texttt{op}}(n) = \mu'_1(n) \cup \mu'_2(n)$, where μ'_1 and μ'_2 are the mappings computed by the algorithm in Fgiure 14 when applied to the parallel interaction graph at the program point $\texttt{exit}_{\texttt{op}}\bullet$. The constraint system at a method invocation site m is defined as follows:

$$\Omega_m = \{\omega(n_1) \subseteq \omega(n_2) . n_1 \in \mu^m_{\texttt{op}}(n_2)\} \cup \bigcup_{\texttt{op} \in \text{callees}(m)} \Omega_{\texttt{op}}$$

The constraint system is considered to be final for a node n if it completely summarizes all of the inside nodes that n can represent during the analysis of the method invocation site. To determine if the constraint system is final for n, the algorithm checks to see that all of the outside nodes that n represented during the analysis have been completely resolved to inside nodes. Formally,

$$\text{final}(m, n) = \forall n' \in \omega_m(n), \texttt{op} \in \text{callees}(m) . \mu^m_{\texttt{op}}(n') \cap N_O = \emptyset$$

For each method `op`, the analysis can choose whether it wishes to compute the interactions between threads to resolve outside nodes. The advantage of doing so is a potential increase in the precision; the disadvantage is a potential increase in the analysis time. If the analysis does not compute the interactions between threads, $\Omega_{\texttt{op}}$ and final(`op`, n) are defined as follows:

$$\begin{array}{rcl} \Omega_{\texttt{op}} & = & \Omega_0 \cup \bigcup_{m \in \text{invocations}(\texttt{op})} \Omega_m \\ \text{final}(\texttt{op}, n) & = & \forall m \in \text{invocations}(\texttt{op}) . \text{final}(m, n) \end{array}$$

Here invocations(`op`) is the set of all method invocation sites in `op`. If the analysis does compute the interactions, $\Omega_{\texttt{op}}$ and final(`op`, n) are defined as follows:

$$\begin{array}{rcl} \Omega_{\texttt{op}} & = & \{\omega(n_1) \subseteq \omega(n_2) . n_1 \in \mu_{\texttt{op}}(n_2)\} \cup \\ & & \Omega_0 \cup \bigcup_{m \in \text{invocations}(\texttt{op})} \Omega_m \\ \text{final}(\texttt{op}, n) & = & \forall n' \in \omega_{\texttt{op}}(n) . \mu_{\texttt{op}}(n') \cap N_O = \emptyset \end{array}$$

The analysis can mix and match these two approaches on a per-method basis; an appropriate policy is to compute the interactions only for methods that start threads.

If a node is final in a given constraint system, the analysis has determined all of the inside nodes that it represents during the analysis of the corresponding method invocation site or method. More formally, if final(m, n), then $\omega_m(n) \cap N_I$ is the set of inside nodes represented by n during the analysis of the method invocation site m. Similarly, if final(`op`, n), then $\omega_{\texttt{op}}(n) \cap N_I$ is the set of inside nodes represented by n during the analysis of the method invocation site `op`.

13 Abstraction Relation

In this section, we characterize the correspondence between parallel interaction graphs and the objects and references created during the execution of the program. A key property of this correspondence is that a single concrete object in the execution of the program may be represented by multiple nodes in the parallel interaction graph. We therefore state the properties that characterize the correspondence using an *abstraction relation*, which relates each object to all of the nodes that represent it.

As the program executes, it creates a set of concrete objects $o \in \mathrm{C}$ and a set of references $r \in R \subseteq (\mathtt{V} \times \mathrm{C}) \cup ((\mathrm{C} \times \mathtt{F}) \times \mathrm{C})$ between objects. At each point in the execution of the program, it is possible to define the following sets of references and objects:

- R_C is the set of references created by the current execution of the current method and all of the analyzed methods that it invokes.
- R_R is the set of references read by the current execution of the current method and all of the analyzed methods that it invokes.
- C_R is the set of objects reachable from the local variables, static class variables, and parameters by following references in $R_C \cup R_R$.
- $R_I = R_C \cap ((\mathrm{C}_R \times F \times \mathrm{C}_R) \cup (\mathtt{V} \times \mathrm{C}_R))$ is the set of inside references. These are the references represented by the set of inside edges in the analysis.
- $R_O = (R_R \cap (\mathrm{C}_R \times F \times \mathrm{C}_R)) - R_I$ is the set of outside references. These are the references represented by the set of outside edges in the analysis.

It is always possible to construct an abstraction relation $\rho \subseteq \mathrm{C} \times N$ between the objects and the nodes in the parallel interaction graph $\langle\langle O, I, e, r\rangle, \tau, \alpha, \pi\rangle$ at the current program point. This relation relates each object to all of the nodes in the points-to escape graph that represent the object during the analysis of the method. The abstraction relation has all of the properties described below.

- Reachable objects are represented by their allocation sites. If o was created at an object creation site within the current execution of the current method or analyzed methods that it invokes, and o is reachable (i.e. $o \in \mathrm{C}_R$), $n \in \rho(o)$, where n is the object creation site's inside node.
- Each object is represented by at most one inside node:
  - $n_1, n_2 \in \rho(o)$ and $n_1, n_2 \in N_I$ implies $n_1 = n_2$
- All outside references have a corresponding outside edge in the points-to escape graph:
  - $\langle\langle \mathtt{cl}, \mathtt{f}\rangle, o\rangle \in R_O$ implies $O(\mathtt{cl}, \mathtt{f}) \cap \rho(o) \neq \emptyset$
  - $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \in R_O$ implies $(\rho(o_1) \times \{\mathtt{f}\}) \times \rho(o_2) \cap O \neq \emptyset$
- All inside references have a corresponding inside edge in the points-to escape graph:
  - $\langle \mathtt{v}, o\rangle \in R_I$ implies $I(\mathtt{v}) \cap \rho(o) \neq \emptyset$
  - $\langle\langle \mathtt{cl}, \mathtt{f}\rangle, o\rangle \in R_I$ implies $I(\mathtt{cl}, \mathtt{f}) \cap \rho(o) \neq \emptyset$
  - $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \in R_I$ implies $(\rho(o_1) \times \{\mathtt{f}\}) \times \rho(o_2) \cap I \neq \emptyset$
- If an object is represented by a captured node, it is represented by only that node:
  - $n \in \rho(o)$ and captured$(\langle O, I, e, r\rangle, n)$ implies $\rho(o) = \{n\}$

  Given this property, we define that an object is captured if it is represented by a captured node. All references to captured objects are either from local variables or from other captured objects:

  - $n \in \rho(o)$, captured$(\langle O, I, e, r\rangle, n)$, and $\langle \mathtt{v}, o\rangle \in R$ implies $\mathtt{v} \in \mathtt{L}$
  - $n_2 \in \rho(o_2)$, captured$(\langle O, I, e, r\rangle, n_2)$, and $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \in R$ implies $\exists n_1 \in N.\rho(o_1) = \{n_1\}$ and captured$(\langle O, I, e, r\rangle, n_1)$

  These properties ensure that captured objects are reachable only via paths that start with the local variables. If an object is captured at a method exit point, it will therefore become inaccessible as soon as the method returns.
- The points-to information in the points-to escape graph completely characterizes the references between objects represented by captured nodes:
  - captured$(\langle O, I, e, r\rangle, n_1)$, captured$(\langle O, I, e, r\rangle, n_2)$, $n_1 \in \rho(o_1), n_2 \in \rho(o_2)$ and $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \notin I$ implies $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \notin R$

14 Experimental Results

We have implemented a combined pointer and escape analysis based on the algorithm described in this paper. We implemented the analysis in the compiler for the Jalapeño JVM [3], a Java virtual machine written in Java with a few unsafe extensions for performing low-level system operations such as explicit memory management and pointer manipulation.

The analysis is implemented as a separate phase of the Jalapeño dynamic compiler, which operates on the Jalapeño intermediate representation. To analyze a class, the algorithm loads the class, converts its methods into the intermediate representation, then analyzes the methods. The final analysis results for the methods are written out to a file. This approach provides excellent support for dynamically loaded programs. It allows the compiler to analyze a large, commonly used package such as the Java Class Libraries once, then reuse the analyze results every time a program is loaded that uses the package. It also supports the delivery of preanalyzed packages. Instead of requiring the analysis to be performed when the package is first loaded into a customer's virtual machine, a vendor could perform the analysis as part of the release process, then ship the analysis results along with the code.

Our benchmark set includes four programs: javac (Java compiler), javacup (parser generator), server (a simple multithreaded web server), and work (a compute benchmark with multiple worker threads). Figure 17 presents the total number of synchronizations required to execute each program. We report counts for three different optimization levels:

- **Original:** No analysis is performed.

- **Interprocedural:** The compiler uses the interprocedural, single-threaded analysis results as defined in Section 9. At the end of each method, it finds all captured nodes and removes all synchronization on the corresponding objects from the counts.
- **Interthread:** The compiler uses the inter-thread analysis as defined in Section 10. In addition to the Interprocedural optimization described above, the compiler uses the thread interaction results. At the end of each method that starts a thread, and at the end of the `main` method, it resolves the interactions between started parallel threads. If all of the synchronization actions on a captured node in the resulting parallel interaction graph are independent, the analysis removes all synchronization on the corresponding objects from the counts.

Application	Original	Interprocedural	Interthread
javac	2,080,116	1,348,814	51,164
javacup	1,704,563	537,040	121,798
server	7,091	6,123	1,842
work	21,877	21,317	2,983

Figure 17: Total Number of Synchronization Operations

For javac and javacup, the inter-thread optimization removes over 92% of the total synchronizations. For server and work, the inter-thread synchronization removes over 74% of the synchronizations. In all of the cases, the Interthread optimization significantly reduces the number of synchronizations as compared to the Interprocedural optimization. To put these results in perspective, recent research with escape analysis algorithms less powerful than our Interprocedural optimization level reported significant speedups from synchronization elimination for a range of Java programs [12, 3].

15 Related Work

In this section, we discuss several areas of related work: pointer analysis, escape analysis, and synchronization optimizations.

15.1 Pointer Analysis for Multithreaded Programs

There have been, to our knowledge, two previously published flow-sensitive pointer analysis algorithms for multithreaded programs. Rugina and Rinard published an algorithm for programs with structured, fork-join parallelism [37]. The algorithm is interprocedural, context-sensitive, and top-down, generating calling contexts in a top-down manner starting with the main method. Each procedure is reanalyzed for each new calling context. Corbett published a algorithm for multithreaded programs that consist of a single procedure [16]. Both analyses use an iterative, fixed-point algorithm to compute the interactions between parallel threads, and must analyze the entire program. Our algorithm, on the other hand, is a bottom-up, compositional, interprocedural algorithm that analyzes each method once to derive a parameterized analysis result that can be specialized for use at all call sites that invoke the method.[3] Unlike Corbett's algorithm, it handles multiple procedures and recursively generated concurrency. Unlike Rugina and Rinard's algorithm, it handles programs with unstructured multithreading.

Rugina and Rinard's algorithm propagates information with three sets of edges: the current set of edges C, interference edges I from other threads, and the set of edges E created by the current thread. Separating C and E enables the algorithm to perform strong updates to shared variables. Strong updates eliminate edges from C, leaving them in E to be correctly observed by other threads. There are two ways to extend the algorithm presented in this paper to handle strong updates to heap allocated objects. The first is to allow the analysis to perform strong updates to captured objects [41]. The second is to adopt the Rugina and Rinard solution and split the set of inside edges in the parallel interaction graphs into a set of current edges and a set of edges created by the current thread, with strong updates removing edges from the set of current edges but leaving them in place in the set of edges created by the current thread.

The interference edges in Rugina and Rinard's analysis correspond to inside edges from parallel threads in the analysis presented in this paper. In general, the set of interference edges coming into a current thread depends on interactions between that current thread and threads that run in parallel with it. Rugina and Rinard's analysis uses a fixed-point algorithm to resolve these interactions and compute a complete set of interference edges for the analysis of each thread. This algorithm repeatedly reanalyzes threads until the sets of interference edges from parallel threads do not change. The analysis presented in this paper takes a different approach. It uses outside edges and nodes to represent all potential interactions of the current thread with other parallel threads. These outside edges and nodes enable the analysis to conceptually derive a complete set of interference edges from parallel threads without a fixed-point algorithm. Instead, the analysis matches inside and outside edges to compute the interactions without reanalyzing each thread.

In general, the analysis of multithreaded programs is a relatively unexplored field. There is an awareness that multithreading significantly complicates program analysis [31], but a full range of standard techniques have yet to emerge. Grunwald and Srinivasan present a dataflow analysis framework for reaching definitions for explicitly parallel programs [25], and Knoop, Steffen and Vollmer present an efficient dataflow analysis framework for bit-vector problems such as liveness, reachability and available expressions, but neither framework applies to pointer analysis [29]. In fact, the application of these frameworks for programs with pointers would require pointer analysis information. Zhu and Hendren present a set of communication optimizations for parallel programs that use information from their pointer analysis; this analysis uses a flow-insensitive analysis to detect pointer variable interference between parallel threads [44]. Hicks also has developed a flow-insensitive analysis specifically for a multithreaded language [27].

15.2 Escape Analysis for Multithreaded Programs

There have been, to our knowledge, four previously published escape analysis algorithms for multithreaded programs [41, 10, 12, 15]. All of these algorithms use the escape information for stack allocation and synchronization elimination. They only analyze single threads, and are designed to find objects that are accessible to only the current thread. If an object escapes the current thread, either to another thread

[3] Recursive methods require an iterative algorithm that may analyze methods multiple times to reach a fixed point.

or by being written into a static class variable, it is marked as globally escaping, and there is no attempt to recapture the object by analyzing the interactions between the threads that access the object. These algorithms are therefore fundamentally sequential program analyses that have been adjusted to ensure that they operate conservatively in the presence of parallel threads. The algorithm presented in this paper, on the other hand, is designed to analyze the interactions between parallel threads. Unlike all other previously published algorithms, it is capable of extracting precise escape information even for objects that are accessible to multiple threads.

15.3 Pointer Analysis for Sequential Programs

Pointer analysis for sequential programs is a relatively mature field. Flow-insensitive analyses, as the name suggests, do not take statement ordering into account, and often use an analysis based on some form of set inclusion constraints to produce a single points-to graph that is valid across the entire program [4, 40, 39]. Many flow-insensitive algorithms scale well to very large programs, in part because they generate one analysis result instead of one per program point and in part because of highly optimized implementations of the inclusion constraint solution algorithms [24]. Because flow-insensitive analyses are insensitive to the order in which statements execute, they model all interleavings and extend trivially to multithreaded programs. Like many flow insensitive algorithms, we use set inclusion constraints as a fundamental tool in our analysis. A difference is that our analysis uses these constraints to formally specify the result of interactions between parallel interaction graphs during the interprocedural and inter-thread analyses, with each interaction generating its own constraint solution problem. The intraprocedural analysis uses a standard fixed-point dataflow approach. Flow-insensitive analyses typically formulate the entire analysis problem as a single collection of set inclusion constraints.

Flow-sensitive analyses take the statement ordering into account, typically using a dataflow analysis to produce a points-to graph or set of alias pairs for each program point [38, 35, 42, 23, 14, 30]. One approach analyzes the program in a top-down fashion starting from the `main` procedure, reanalyzing each potentially invoked procedure in each new calling context [42, 23]. Another approach analyzes the program in a bottom-up fashion, extracting a single analysis result for each procedure. The result is reused at each call site that may invoke the procedure [38, 13]. Our algorithm builds on these previous approaches. It is extended to include escape and action ordering information and to explicitly represent potential interactions using outside edges and nodes. These extensions enable the algorithm to generalize in a straightforward way to model interactions between parallel threads.

Multithreading introduces one particulary subtle point. Consider a load of a reference from an inside node, which represents an object created within the computation of the currently analyzed method. In a sequential program, the load would always return a reference created within the analysis of the current method — because the inside node did not exist before the method was invoked, no unanalyzed code could have executed to write a reference into the object. But in a multithreaded program, an unanalyzed parallel thread may write references into an object as soon as it escapes. The analysis for multithreaded programs must therefore assume that every load from an escaped object may access a reference created outside the current analysis scope. Our analysis deals with this possibility by using outside edges and outside nodes to represent the results of loads from escaped objects.

There is a similarity between the outside nodes in our analysis and invisible variables in previous analyses [30, 23, 42]. Both outside nodes and invisible variables are used to represent objects from outside the analysis context during the analysis of a method or procedure. One difference is that when the analysis generates contexts in a top-down fashion, it has a complete characterization of the aliasing and points-to relationships involving all invisible variables. In this context, invisible variables primarily serve to enable the analysis to reuse analysis results for contexts with the same aliasing and points-to relationships between procedure parameters. Because our analysis is bottom-up, it knows nothing about the relationships involving objects represented by outside nodes. It therefore analyzes each method under the two assumptions that there are no aliases between outside nodes, and that every load from an escaped node may access a reference created outside the method. One implication of this difference is that different invisible variables always represent disjoint sets of objects, while different outside nodes may represent overlapping sets of objects.

Invisible variables and outside nodes also support a particular kind of precision in the analysis. Consider a method invoked at multiple call sites. It may be the case that an object is allocated inside the method, escapes the method, but is recaptured at each call site. In this case, the use of invisible variables or outside nodes enables the analysis to recognize that the object was recaptured. It can therefore separate the different instantiations of the allocated object from each other in the analysis. To our knowledge, the only published analyses that can separate the different instantiations of the object are both flow sensitive and use some variant of the concept of invisible variables [30, 23, 42]. Context-insensitive analyses simply merge the information from the different call sites. In the absence of recursion, other context-sensitive analyses are capable of separating the different instantiations [32], but merge information from recursive call sites in a way that destroys the distinction between multiple instantiations of the same variable in a recursive procedure [32]. A flow-insensitive, constraint-based analysis with polymorphic recursion may be able to separate the instantiations and recover this particular kind of precision.

15.4 Escape Analysis

There has been a fair amount of work on escape analysis in the context of functional languages [7, 5, 43, 18, 19, 9, 26]. The implementations of functional languages create many objects (for example, cons cells and closures) implicitly. These objects are usually allocated in the heap and reclaimed later by the garbage collector. It is often possible to use a lifetime or escape analysis to deduce bounds on the lifetimes of these dynamically created objects, and to perform optimizations to improve their memory management.

Deutsch [18] describes a lifetime and sharing analysis for higher-order functional languages. His analysis first translates a higher-order functional program into a sequence of operations in a low-level operational model, then performs an analysis on the translated program to determine the lifetimes of dynamically created objects. The analysis is a whole-program analysis. Park and Goldberg [5] also describe an escape analysis for higher-order functional languages. Their analysis is less precise than Deutsch's. It is, however, conceptually simpler and more efficient. Their main contribution was to extend escape analysis to include lists. Deutsch [19] later presented an analysis that extracts the same information but runs in almost linear time. Blanchet [9] extended this algorithm to work in the presence of imperative features and polymorphism. He also provides a correctness proof and some experimental results.

Baker [7] describes an novel approach to higher-order escape analysis of functional languages based on the type inference (unification) technique. The analysis provides escape information for lists only. Hannan also describes a type-based analysis in [26]. He uses annotated types to describe the escape information. He only gives inference rules and no algorithm to compute annotated types.

15.5 Synchronization Optimizations

Diniz and Rinard [20, 21] describe several algorithms for performing synchronization optimizations in parallel programs. The basic idea is to drive down the locking overhead by coalescing multiple critical sections that acquire and release the same lock multiple times into a single critical section that acquires and releases the lock only once. When possible, the algorithm also coarsens the lock granularity by using locks in enclosing objects to synchronize operations on nested objects. Plevyak and Chien describe similar algorithms for reducing synchronization overhead in sequential executions of concurrent object-oriented programs [34].

Several research groups have recently developed synchronization optimization techniques for Java programs. Aldrich, Chambers, Sirer, and Eggers describe several techniques for reducing synchronization overhead, including synchronization elimination for thread-private objects and several optimizations that eliminate synchronization from nested monitor calls [2]. Blanchet describes a pure escape analysis based on an abstraction of a type-based analysis [10]. The implementation uses the results to eliminate synchronization for thread-private objects and to allocate captured objects on the stack. Bogda and Hoelzle describe a flow-insensitive escape analysis based on global set inclusion constraints [12]. The implementation uses the results to eliminate synchronization for thread-private objects. A limitation is that the analysis is not designed to find captured objects that are reachable via paths with more than two references.

Choi, Gupta, Serrano, Sreedhar, and Midkiff present a compositional dataflow analysis for computing reachability information [15]. The analysis results are used for synchronization elimination and stack allocation of objects. Like the analysis presented in this paper, it uses an extension of points-to graphs with abstract nodes that may represent multiple objects. It does not distinguish between inside and outside edges, but does contain an optimization, deferred edges, that is designed to improve the efficiency of the analysis. The approach classifies objects as globally escaping, escaping via an argument, and not escaping. Because the primary goal was to compute escape information, the analysis collapses globally escaping subgraphs into a single node instead of maintaining the extracted points-to information. Our analysis retains this information, which is crucial for developing a pointer analysis algorithm that takes interactions between threads into account.

16 Conclusion

This paper presents a new combined pointer and escape analysis algorithm for unstructured multithreaded programs. It extends the current state of the art in two ways: it is the first interprocedural, flow-sensitive pointer analysis algorithm for unstructured multithreaded programs, and it is the first algorithm to extract precise escape analysis information for objects accessible to multiple threads. We have implemented the algorithm in the IBM Jalapeño virtual machine, and used the analysis results to perform a synchronization elimination optimization. Our experimental results show that, for our set of benchmark applications, the analysis can successfully remove between 75% and 95% of the total synchronizations.

In the long run, we believe the most important concept in this research may turn out to be designing analysis algorithms from the perspective of extracting and representing interactions between analyzed and unanalyzed regions of the program. This approach leads to clean, compositional algorithms that are capable of analyzing arbitrary parts of complete or incomplete programs.

References

[1] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, CA, June 1992.

[2] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Proceedings of the 6th International Static Analysis Symposium*, September 1999.

[3] B. Alpern, D. Attanasio, A. Cochi, D. Lieber, S. Smith, T. Ngo, and J. Barton. Implementing jalapeño in java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[5] B. Goldberg and Y. Park. Escape analysis on lists. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 116–127, July 1992.

[6] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1999.

[7] H. Baker. Unifying and conquer (garbage, updating, aliasing ...) in functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 218–226, 1990.

[8] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A compiler-managed memory system for Raw machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[9] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1998. ACM, ACM, New York.

[10] 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, November 1999.

[11] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995. ACM, New York.

[12] 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, November 1999.

[13] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1999.

[14] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC, January 1993. ACM.

[15] 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, November 1999.

[16] J. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, March 1998.

[17] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[18] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Programming Languages*, pages 157–168, San Francisco, CA, January 1990. ACM, ACM, New York.

[19] A. Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997. ACM, ACM, New York.

[20] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, pages 285–299, San Jose, CA, August 1996. Springer-Verlag.

[21] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 187–200, Paris, France, January 1997. ACM, New York.

[22] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):2218–244, March 1998.

[23] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994. ACM, New York.

[24] M. Fahndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[25] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[26] J. Hannan. A type-based analysis for block allocation in functional languages. In *Proceedings of the Second International Static Analysis Symposium*. ACM, ACM, New York, September 1995.

[27] J. Hicks. Experiences with compiler-directed storage reclamation. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 95–105, June 1993.

[28] S. Horowitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[29] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[30] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[31] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–105–113, 1990.

[32] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *1997 International Conference on Software Engineering*, Boston, MA, May 1997.

[33] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*. ACM, January 1995.

[34] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*, San Francisco, CA, January 1995. ACM, New York.

[35] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[36] R. Rugina and M. Rinard. Symbolic analysis of divide and conquer programs. In *Submitted to PLDI '00.*

[37] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.

[38] P. Sathyanathan and M. Lam. Context-sensitive interprocedural pointer analysis in the presence of dynamic aliasing. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Springer-Verlag.

[39] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997.

[40] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[41] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[42] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.

[43] Y. Tang and P. Jouvelot. Control-flow effects for escape analysis. In *Workshop on Static Analysis*, pages 313–321, September 1992.

[44] H. Zhu and L. Hendren. Communication optimizations for parallel C programs. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.