Pointer and Escape Analysis for Multithreaded Programs *

Alexandru Salcianu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
salcianu@lcs.mit.edu

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

ABSTRACT

This paper presents a new combined pointer and escape analysis for multithreaded programs. The algorithm uses a new abstraction called *parallel interaction graphs* to analyze the interactions between threads and extract precise points-to, escape, and action ordering information for objects accessed by multiple threads. The analysis is compositional, analyzing each method or thread once to extract a parameterized analysis result that can be specialized for use in any context. It is also capable of analyzing programs that use the unstructured form of multithreading present in languages such as Java and standard threads packages such as POSIX threads.

We have implemented the analysis in the MIT Flex compiler for Java and used the extracted information to 1) verify that programs correctly use region-based allocation constructs, 2) eliminate dynamic checks associated with the use of regions, and 3) eliminate unnecessary synchronization. Our experimental results show that analyzing the interactions between threads significantly increases the effectiveness of the region analysis and region check elimination, but has little effect for synchronization elimination.

1. INTRODUCTION

Multithreading is a key structuring technique for modern software. Programmers use multiple threads of control for many reasons: to build responsive servers that communicate with multiple parallel clients [15], to exploit the parallelism in shared-memory multiprocessors [5], to produce sophisticated user interfaces [16], and to enable a variety of other program structuring approaches [11].

Research in program analysis has traditionally focused on sequential programs [14]; extensions for multithreaded programs have usually assumed a block structured, parbegin/parend form of multithreading in which a parent thread starts several parallel threads, then immediately blocks waiting for them to finish [12, 19]. But the standard form of multithreading supported by languages such as Java and threads packages such as POSIX threads is unstructured — child threads execute independently of their parent threads. The software structuring techniques described above are designed to work with this form of multithreading, as are many recommended design patterns [13]. But because the lifetimes of child threads potentially exceed the lifetime of their starting procedure, unstructured multithreading significantly complicates the interprocedural analysis of multithreaded programs.

1.1 Analysis Algorithm

This paper presents a new combined pointer and escape analysis for multithreaded programs, including programs with unstructured forms of multithreading. The algorithm is based on a new abstraction, *parallel interaction graphs*, which maintain precise points-to, escape, and action ordering information for objects accessed by multiple threads. Unlike previous escape analysis abstractions, parallel interaction graphs enable the algorithm to analyze the interactions between parallel threads. The analysis can therefore capture objects that are accessed by multiple threads but do not escape a given multithreaded computation. It can also fully characterize the points-to relationships for objects accessed by multiple parallel threads.

Because parallel interaction graphs characterize all of the potential interactions of the analyzed method or thread with its callers and other parallel threads, the resulting analysis is compositional at both the method and thread levels — it analyzes each method or thread once to produce a single general analysis result that can be specialized for use in any context.[1] Finally, the combination of points-to and escape information in the same abstraction enables the algorithm to analyze only part of the program, with the analysis result becoming more precise as more of the program is analyzed.

1.2 Application to Region-Based Allocation

We have implemented our analysis in the MIT Flex compiler for Java. The information that it produces has many potential applications in compiler optimizations, software engineering, and as a foundation for further program analysis. This paper presents our experience using the analysis to optimize and check safety conditions for programs that use region-based allocation constructs instead of relying on garbage collection. Region-based allocation allows the program to run (a potentially multithreaded) computation in

*The research was supported in part by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-63513.

PPoPP'01, June 18-20, 2001, Snowbird, Utah, USA.

[1] Recursive methods or recursively generated threads may require an iterative algorithm that may analyze methods or threads in the same strongly connected component multiple times to reach a fixed point.

the context of a specific allocation region. All objects created by the computation are allocated in the region and deallocated when the computation finishes. To avoid dangling references, the implementation must ensure that the objects in the region do not outlive the associated computation. One standard way to achieve this goal is to dynamically check that the program never attempts to create a reference from one object to another object allocated in a region with a shorter lifetime [4]. If the program does attempt to create such a reference, the implementation refuses to create the reference and throws an exception. Unfortunately, this approach imposes dynamic checking overhead and introduces a new failure mode for programs that use region-based allocation.

We have used our analysis to statically verify that our multithreaded benchmark programs use region-based allocation correctly. It therefore provides a safety guarantee to the programmer and enables the compiler to eliminate the dynamic region reference checks. We also found that intrathread analysis alone is not powerful enough — the algorithm must analyze the interactions between parallel threads to verify the correct use of region-based allocation.

We also used our analysis for the more traditional purpose of synchronization elimination. While our algorithm is quite effective at enabling this optimization, for our multithreaded benchmarks, the interthread analysis provides little additional benefit over the standard intrathread analysis.

1.3 Contributions

This paper makes the following contributions:

- **Abstraction:** It presents a new abstraction, parallel interaction graphs, for the combined pointer and escape analysis of programs with unstructured multithreading.
- **Analysis:** It presents a new algorithm for analyzing multithreaded programs. The algorithm is compositional and analyzes interactions between parallel threads.
- **Region-Based Allocation:** It presents our experience using the analysis to statically verify that programs correctly use region-based allocation constructs. The benefits include providing a safety guarantee for the program and eliminating the overhead of dynamic region reference checks.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates how the algorithm works. Section 3 presents the abstractions that the analysis uses, while Section 4 presents the analysis algorithm and Section 5 discusses the analysis uses. We discuss experimental results in Section 6, related work in Section 7, and conclude in Section 8.

2. EXAMPLE

We next present a simple example that illustrates how the analysis works.

2.1 Structure of the Parallel Computation

Figure 1 presents a multithreaded Java program that computes the Fibonacci number of its input. The `Task` class implements a parallel divide and conquer algorithm for this computation. Each `Task` stores an `Integer` object in its `source` field as input and produces a new `Integer` object in its `target` field as output.[2]

This program illustrates several common patterns for multithreaded programs. First, it uses threads to implement parallel computations. Second, when a thread starts its execution, it points to objects that hold the input data for its computation. Finally, when the computation finishes, it writes references to its result objects into its thread object for the parent computation to read.

```
class main {
  public static void main(String args[]) {
    int i = Integer.parseInt(args[0]);
    Fib f = new Fib(i);
    Region r = new Region();
    r.enter(f);
  }
}
class Fib implements Runnable {
  int source;

  Fib(int i) { source = i; }

  public void run() {
    Task t = new Task(new Integer(source));
    t.start();
    try {
      t.join();
    } catch (Exception e) { System.out.println(e); }
    System.out.println(t.target.toString());
  }
}
class Task extends Thread {
  public Integer source;
  public Integer target;

  Task(Integer s) { source = s; }

  public void run() {
    int v = source.intValue();
    if (v <= 1) {
      target = source;
    } else {
      Task t1 = new Task(new Integer(v-1));
      Task t2 = new Task(new Integer(v-2));
      t1.start();
      t2.start();
      try {
        t1.join();
        t2.join();
      } catch (Exception e) { System.out.println(e); }
      int x = t1.target.intValue();
      int y = t2.target.intValue();
      target = new Integer(x + y);
    }
  }
}
```

Figure 1: Multithreaded Fibonacci Example

2.2 Regions and Memory Management

As the computation runs, it continually allocates new `Task` objects for the parallel subcomputations and new `Integer` objects to hold their inputs and outputs. The lifetimes of

[2]This program uses the standard Java thread creation mechanism. The statement `t1.start()` creates a new parallel thread of control. This new thread of control then invokes the `run` method of the `Task` class on the `t1` object. This `start/run` linkage is the standard way to execute new threads in Java.

these objects are contained in the lifetime of the Fibonacci computation, and die when this computation finishes. A standard memory management system would not exploit this property. The `Task` and `Integer` objects would be allocated out of the garbage-collected heap, increasing the memory consumption rate, the garbage collection frequency, and therefore the garbage collection overhead.

Region-based allocation provides an attractive alternative. Instead of allocating all objects out of a single garbage-collected heap, region-based approaches allow the program to create multiple memory regions, then allocate each object in a specific region. When the program no longer needs any of the objects in the region, it deallocates all of the objects in that region without garbage collection.

Researchers have proposed many different region-based allocation systems. Our example (and our implemented system) uses the approach standardized in the Real-Time Java specification [4]. Before the `main` program invokes the Fibonacci computation, it creates a new memory region `r`. The statement `r.enter(f)` executes the `run` method of the `f` object (and all of the methods or threads that it executes) in the context of the new region `r`. When one of the threads in this computation creates a new object, the object is allocated in the region `r`. When the entire multithreaded computation terminates, all of the objects in the region are deallocated without garbage collection. The `Task` and `Integer` objects are therefore managed independently of the garbage collected heap and do not increase the garbage collection frequency or overhead. Region-based allocation is an attractive alternative to garbage collection because it exploits the correspondence between the lifetimes of objects and the lifetimes of computations to deliver a more efficient memory management mechanism.

2.3 Regions and Dangling Reference Checks

One potential problem with region-based allocation is the possibility of dangling references. If an object whose lifetime exceeds the region's lifetime refers to an object allocated inside the region, any use of the reference after the region is deallocated will access potentially recycled garbage, violating the memory safety of the program. The Real-Time Java specification eliminates this possibility as follows. It allows the computation to create a hierarchy of nested regions and ensures that no parent region is deallocated before one of its child regions. Each region is associated with a (potentially multithreaded) computation; the objects in the region are deallocated when its computation terminates and the objects in all of its child regions have been deallocated. The implementation dynamically checks all assignments to object fields to ensure that the program never attempts to create a reference that goes down the hierarchy from an object in an ancestor region to an object in a child region. If the program does attempt to create such a reference, the check fails. The implementation prevents the assignment from taking place and throws an exception.

While these checks ensure the memory safety of the execution, they impose additional execution time overhead and introduce a new failure mode for the software. Our goal is to analyze the program and statically verify that the checks never fail. Such an analysis would enable the compiler to eliminate all of the dynamic region checks. It would also provide the programmer with a guarantee that the program would never throw an exception because a check failed.

2.4 Analysis in the Example

We use a generalized escape analysis to determine whether any object allocated in a given region escapes the computation associated with the region. If none of the objects escape, the program will never attempt to create a dangling reference and the compiler can eliminate all of the checks. The algorithm first performs an intrathread, interprocedural analysis to derive a parallel interaction graph at the end of each method. Figures 2 and 3 present the analysis results for the `run` methods in the `Fib` and `Task` classes, respectively.

2.4.1 Points-to Graphs

The first component of the parallel interaction graph is the points-to graph. The nodes in this graph represent objects; the edges represent references between objects. There are two kinds of edges: *inside* edges, which represent references created within the analyzed part of the program (for Figure 2, the sequential computation of the `Fib.run` method), and *outside* edges, which represent references read from objects potentially accessed outside the analyzed part of the program. In our figures, solid lines denote inside edges and dashed lines denote outside edges.

There are also several kinds of nodes. *Inside* nodes represent objects created within the analyzed part of the program. There is one inside node for each object creation site in the program; that node represents all objects created at that site. *Parameter* nodes represent objects passed as parameters to the currently analyzed method; *load* nodes represent objects accessed by reading a reference in an object potentially accessed outside the analyzed part of the program. Together, the parameter and load nodes make up the set of *outside* nodes. In our figures, solid circles denote inside nodes and dashed circles denote outside nodes.

In Figure 2, nodes `1` and `4` are outside nodes. Node `1` represents the `this` parameter of the method, while node `4` represents the object whose reference is loaded by the expression `t.target` at line 2 of the example at the end of the `Fib.run` method. Nodes `2` and `3` are inside nodes, and denote the `Task` and `Integer` objects created in the statement `Task t = new Task(new Integer(source))` at line `1` of the example.

2.4.2 Started Thread Information

The parallel interaction graph contains information about which threads were started by the analyzed part of the program. In Figure 2, node `2` represents the started `Task` thread that implements the entire Fibonacci computation. In Figure 3, nodes `8` and `11` represent the two threads that implement the parallel subtasks in the computation. The interthread analysis uses the started thread information when it computes the interactions between the current thread and threads that execute in parallel with the current thread.

2.4.3 Escape Information

The parallel interaction graph contains information about how objects escape the analyzed part of the program to be accessed by the unanalyzed part. A node escapes if it is a parameter node or represents an unanalyzed thread started within the analyzed part of the program. It also escapes if it is reachable from an escaped node. In Figure 2, node `1` escapes because it is passed as a parameter, while nodes `3` and `4` escape because they are reachable from the unanalyzed thread node `2`.

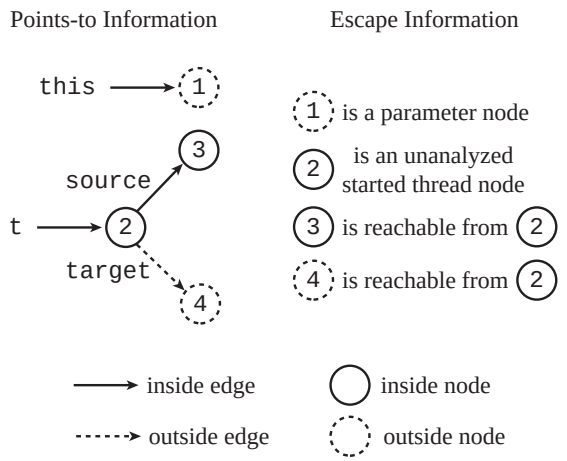

Figure 2: Analysis Result for `Fib.run`

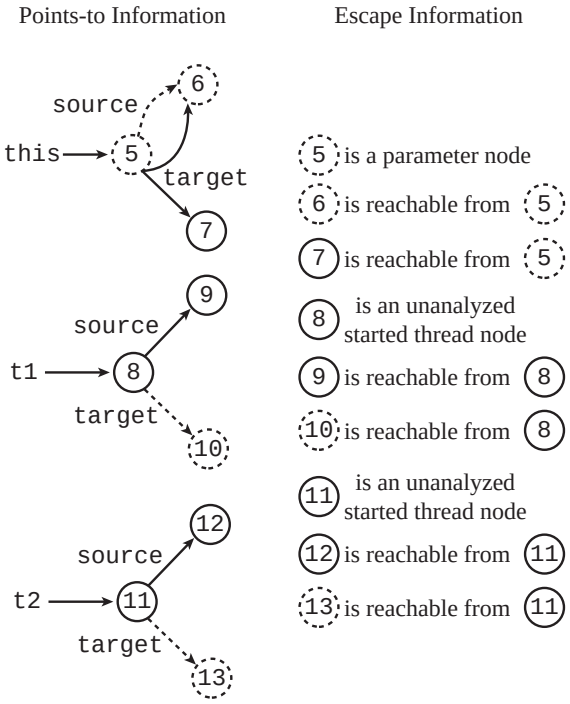

Figure 3: Analysis Result for `Task.run`

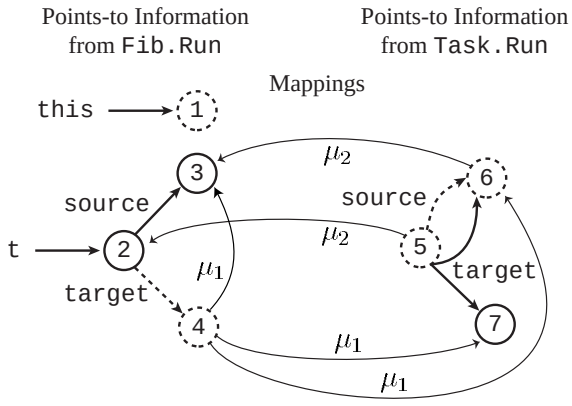

Figure 4: Mappings for Interthread Analysis of `Fib.run` and `Task.run`

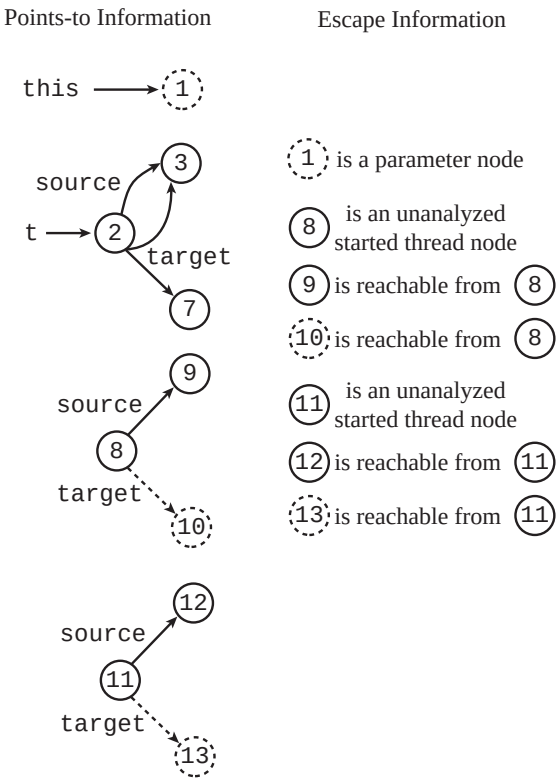

Figure 5: Analysis Result After First Interthread Analysis

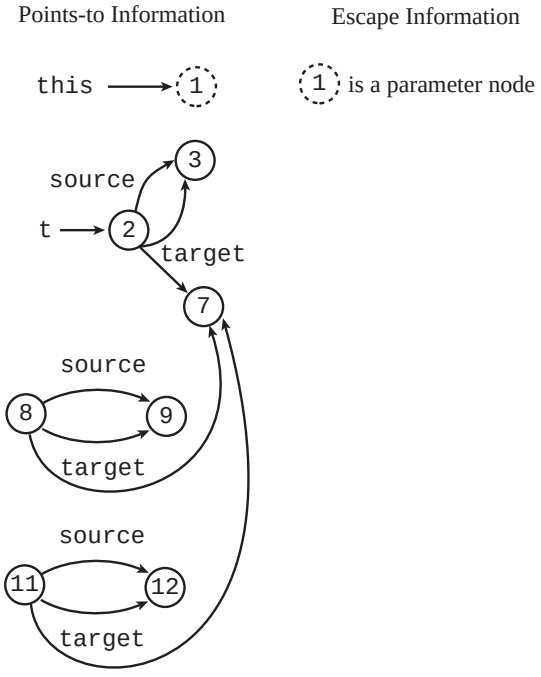

Figure 6: Final Analysis Result for `Fib.run`

2.5 Interthread Analysis

Previously proposed escape analyses treat threads very conservatively — if an object is reachable from a thread object, the analyses assume that it has permanently escaped [2, 3, 6, 22]. Our algorithm, however, analyzes the interactions between threads to recapture objects accessed by multiple threads. The foundation of the interthread analysis is the construction of two mappings μ_1 and μ_2 between the nodes of the parallel interaction graphs of the parent and child threads. Each outside node is mapped to another node if the two nodes represent the same object during the analysis. The mappings are used to combine the parallel interaction graph from the child thread into the parallel interaction graph from the parent thread. The result is a new parallel interaction graph that summarizes the parallel execution of the two threads.

Figure 4 presents the mappings from the interthread analysis of `Fib.run` and the `Task.run` method for the thread that `Fib.run` starts. The algorithm computes these mappings as follows:

- **Initialization:** Inside the `Fib.run` method, node `2` represents the started `Task` thread. Inside the `Task.run` method, node `5` represents the same started thread. The algorithm therefore initializes μ_2 to map node `5` to node `2`.

- **Matching `target` edges:** The analysis of the `Task.run` method creates inside edges from node `5` to nodes `6` and `7`. These edges have the label `target`, and represent references between the corresponding `Task` and `Integer` objects during the execution of the `Task.run` method.

  The `Fib.run` method reads these references to obtain the result of the `Task.run` method. The outside edge from node `2` to node `4` represents these references during the analysis of the `Fib.run` method. The analysis therefore matches the outside edge from the `Fib.run` method (from node `2` to node `4`) against the inside edges from the `Task.run` method to compute that node `4` represents the same objects as nodes `6` and `7`. The result is that μ_1 maps node `4` to nodes `6` and `7`.

- **Matching `source` edges:** The analysis of the `Fib.run` method creates an inside edge from node `2` to node `3`. This edge has the label `source`, and represents a reference between the corresponding `Task` and `Integer` objects during the execution of the `Fib.run` method.

  The `Task.run` method reads this reference to obtain its input. The outside edge from node `5` to node `6` represents this reference during the analysis of the `Task.run` method. The interthread analysis therefore matches the outside edge from the `Task.run` method (from node `5` to node `6`) against the inside edge from the `Fib.run` method (from node `2` to node `3`) to compute that node `6` represents the same objects as node `3`. The result is that μ_2 maps node `6` to node `3`.

- **Transitive Mapping:** Because μ_1 maps node `4` to node `6` and μ_2 maps node `6` to node `3`, the analysis computes that node `4` represents the same object as node `3`. The result is that μ_1 maps node `4` to node `3`.

Note that the matching process models interactions in which one thread reads references created by the other thread. Because the threads execute in parallel, the matching is symmetric.

The analysis uses μ_1 and μ_2 to combine the two parallel interaction graphs and obtain a new graph that represents the combined effect of the two threads. Figure 5 presents this graph, which the analysis computes as follows:

- **Edge Projections:** The analysis projects the edges through the mappings to augment nodes from one parallel interaction graph with edges from the other graph. In our example, the analysis projects the inside edge from node `5` to node `6` through μ_2 to generate new inside edges from node `2` to nodes `3` and `7`. It also generates other edges involving outside nodes, but removes these edges during the simplification step.

- **Graph Combination:** The analysis combines the two graphs, omitting the outside node that represents the `this` parameter of the started thread (node `5` in our example).

- **Simplification:** The analysis removes all outside edges from captured nodes, all outside nodes that are not reachable from a parameter node or unanalyzed started thread node, and all inside nodes that are not reachable from a live variable, parameter node, or unanalyzed started thread node.

In our example, the analysis recaptures the (now analyzed) thread node `2`. Nodes `3` and `7` are also captured *even though they are reachable from a thread node.* The analysis removes nodes `4` and `6` in the new graph because they are not reachable from a parameter node or unanalyzed thread node. Note that because the interactions with the thread nodes `8` and `11` have not yet been analyzed, those nodes and all nodes reachable from them escape.

Because our example program uses recursively generated parallelism, the analysis must perform a fixed point computation during the interthread analysis. Figure 6 presents the final parallel interaction graph from the end of the `Fib.run` method, which is the result of this fixed point analysis. The analysis has recaptured all of the inside nodes, including the task nodes. Because none of the objects represented by these nodes escapes the computation of the `Fib.run` method, its execution in a new region will not violate the region referencing constraints.

3. ANALYSIS ABSTRACTION

We next formally present the abstraction (parallel interaction graphs) that the analysis uses. In addition to the points-to and escape information discussed in Section 2, parallel interaction graphs can also represent ordering information between actions (such as synchronization actions) from parent and child threads. This ordering information enables the analysis to determine when thread start events temporally separate actions of parent and child threads. This information may, for example, enable the analysis to determine that a parent thread performs all of its synchronizations on a given object before a child thread starts its execution and synchronizes on the object. To simplify the presentation, we assume that the program does not use static class variables,

all the methods are analyzable and none of the methods returns a result. Our implemented analysis correctly handles all of these aspects [20].

3.1 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes, which is the disjoint union of the set N_I of inside nodes and the set N_O of outside nodes. The set of thread nodes $N_T \subseteq N_I$ represents thread objects. The set of outside nodes is the disjoint union of the set N_L of load nodes and the set N_P of parameter nodes. There is also a set $\mathtt{f} \in \mathtt{F}$ of fields in objects, a set $\mathtt{v} \in \mathtt{V}$ of local and parameter variables, and a set $\mathtt{l} \in \mathtt{L} \subseteq \mathtt{V}$ of local variables.

3.2 Points-To Escape Graphs

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

- $O \subseteq N \times \mathtt{F} \times N_L$ is a set of outside edges. We use the notation $O(n_1, \mathtt{f}) = \{n_2 | \langle n_1, \mathtt{f}, n_2 \rangle \in O\}$.
- $I \subseteq (N \times \mathtt{F} \times N) \cup (\mathtt{V} \times N)$ is a set of inside edges. We use the notation $I(\mathtt{v}) = \{n | \langle \mathtt{v}, n \rangle \in I\}$, $I(n_1, \mathtt{f}) = \{n_2 | \langle n_1, \mathtt{f}, n_2 \rangle \in I\}$.
- $e : N \rightarrow \mathcal{P}(N)$ is an escape function that records the escape information for each node.[3] A node escapes if it is reachable from a parameter node or from a node that represents an unanalyzed parallel thread.

The escape function must satisfy the invariant that if n_1 points to n_2, then n_2 escapes in at least all of the ways that n_1 escapes. When the analysis adds an edge to the points-to escape graph, it updates the escape function so that it satisfies this invariant. We define the concepts of escaped and captured nodes as follows:

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

3.3 Parallel Interaction Graphs

A parallel interaction graph is a tuple $\langle \langle O, I, e \rangle, \tau, \alpha, \pi \rangle$:

- The thread set $\tau \subseteq N$ represents the set of unanalyzed thread objects started by the analyzed computation.
- The action set α records the set of actions executed by the analyzed computation. Each synchronization action $\langle \mathtt{sync}, n_1, n_2 \rangle \in \alpha$ has a node n_1 that represents the object on which the action was performed and a node n_2 that represents the thread that performed the action. If the action was performed by the current thread, n_2 is the dummy current thread node $n_{\mathrm{CT}} \in N_T$. Our implementation can also record actions such as reading an object, writing an object, or invoking a given method on an object. It is straightforward to generalize the concept of actions to include actions performed on multiple objects.
- The action order π records ordering information between the actions of the current thread and threads that execute in parallel with the current thread.
  - $\langle \langle \mathtt{sync}, n_1, n_2 \rangle, n \rangle \in \pi$ if the synchronization action $\langle \mathtt{sync}, n_1, n_2 \rangle$ may have happened after one of the threads represented by n started executing. In this case, the actions of a thread represented by n may conflict with the action.
  - $\langle \langle n_1, \mathtt{f}, n_2 \rangle, n \rangle \in \pi$ if a reference represented by the outside edge $\langle n_1, \mathtt{f}, n_2 \rangle$ may have been read after one of the threads represented by n started executing. In this case, the outside edge may represent a reference written by a thread represented by n.

We use the notation $\pi @ n = \{a | \langle a, n \rangle \in \pi\}$ to denote the set of actions and outside edges in π that may occur in parallel with a thread represented by n.

[3]Here $\mathcal{P}(N)$ is the set of all subsets of N, so that $e(n)$ is the set of nodes through which n escapes.

4. ANALYSIS ALGORITHM

For each program point, the algorithm computes a parallel interaction graph for the current analysis scope at that point. For the intraprocedural analysis, the analysis scope is the currently analyzed method up to that point. The interprocedural analysis extends the scope to include the (transitively) called methods; the interthread analysis further extends the scope to include the started threads.

We next present the analysis, identifying the program representation, the different phases, and the key algorithms in the interprocedural and interthread phases.

4.1 Program Representation

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

The control flow graph for each method $\mathtt{op}$ starts with an enter statement $\mathtt{enter}_{\mathtt{op}}$ and ends with an exit statement $\mathtt{exit}_{\mathtt{op}}$.

4.2 Intraprocedural Analysis

The intraprocedural analysis is a forward dataflow analysis that propagates parallel interaction graphs through the statements of the method's control flow graph. Each method is analyzed under the assumption that the parameters are *maximally unaliased*, i.e., point to different objects. For a method with formal parameters $\mathtt{v}_0, \ldots, \mathtt{v}_n$, the initial parallel interaction graph at the entry point of the method is $\langle \langle \emptyset, \{\langle \mathtt{v}_i, n_{\mathtt{v}_i} \rangle\}, \lambda n.\text{if } n = n_{\mathtt{v}_i} \text{ then } \{n\} \text{ else } \emptyset \rangle, \emptyset, \emptyset, \emptyset \rangle$, where $n_{\mathtt{v}_i}$ is the parameter node for parameter $\mathtt{v}_i$. If the method is invoked in a context where some of the parameters may point to the same object, the interprocedural analysis described below in Section 4.4 merges parameter nodes to conservatively model the effect of the aliasing.

The transfer function $\langle G', \tau', \alpha', \pi' \rangle = [\![\mathtt{st}]\!](\langle G, \tau, \alpha, \pi \rangle)$ models the effect of each statement $\mathtt{st}$ on the current parallel interaction graph. Figure 7 graphically presents the rules that determine the new points-to graphs for the different basic 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

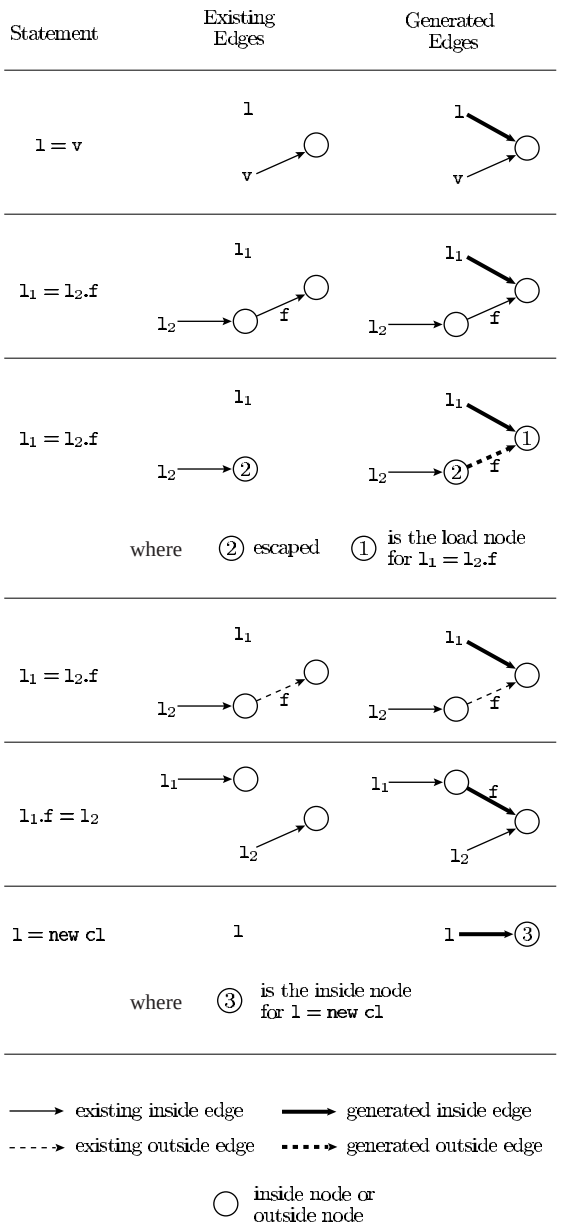

Figure 7: Generated Edges for Basic Statements

$$\tau' = \tau \cup I(\mathtt{l})$$
$$e'(n) = \begin{cases} e(n) \cup \{n'\} & \text{if } n' \in I(\mathtt{l}) \text{ and} \\ & \quad n \text{ is reachable in } O \cup I \text{ from } n' \\ e(n) & \text{otherwise} \end{cases}$$

Figure 8: Transfer Function for `l.start()`

$$\alpha' = \alpha \cup \{\mathtt{sync}\} \times I(\mathtt{l}) \times \{n_{\mathrm{CT}}\}$$
$$\pi' = \pi \cup (\{\mathtt{sync}\} \times I(\mathtt{l}) \times \{n_{\mathrm{CT}}\}) \times \tau$$

Figure 9: Transfer Function for `l.acquire()` and `l.release()`

conditions. The interpretation of each row is that whenever the points-to escape graph contains the existing edges and the side conditions are satisfied, the transfer function for the statement generates the new edges. Assignments to a variable kill existing edges from that variable; assignments to fields of objects leave existing edges in place.

In addition to updating the outside and inside edge sets, the transfer function also updates the the escape function e to ensure that if n_1 points to n_2, then n_2 escapes in at least all of the ways that n_1 escapes. Except for load statements, the transfer functions leave τ, α, and π unchanged. For a load statement $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$ the transfer function updates the action order π to record that any new outside edges may be created in parallel with the threads modeled by the nodes in τ (here n_L is the load node for $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$):

$$\pi' = \pi \cup \{\langle n_1, \mathtt{f}, n_L\rangle | n_1 \in I(\mathtt{l}_2) \wedge \mathrm{escaped}(\langle O, I, e\rangle, n_1)\} \times \tau$$

Figure 8 presents the transfer function for an `l.start()` statement, which adds the started thread nodes to τ and updates the escape function. Figure 9 presents the transfer function for synchronization statements, which add the corresponding synchronization actions into α and record the actions as executing in parallel with all of the nodes in τ. At control-flow merges, the confluence operation takes the union of the inside and outside edges, thread sets, actions, and action orders.

4.3 Mappings

Mappings $\mu : N \to \mathcal{P}(\mathcal{N})$ implement the substitutions that take place when combining parallel interaction graphs. During the interprocedural analysis, for example, a parameter node from a callee is mapped to all of the nodes at the call site that may represent the corresponding actual parameter. Given an analysis component ξ, $\xi[\mu]$ denotes the component after replacing each node n in ξ with $\mu(n)$:[4]

$$\begin{aligned}
\tau[\mu] &= \textstyle\bigcup_{n \in \tau} \mu(n) \\
O[\mu] &= \textstyle\bigcup_{\langle n, \mathtt{f}, n_L\rangle \in O} \mu(n) \times \{\mathtt{f}\} \times \{n_L\} \\
I[\mu] &= \textstyle\bigcup_{\langle n_1, \mathtt{f}, n_2\rangle \in I} \mu(n_1) \times \{\mathtt{f}\} \times \mu(n_2) \cup \bigcup_{\langle \mathtt{v}, n\rangle \in I} \{\mathtt{v}\} \times \mu(n) \\
\alpha[\mu] &= \textstyle\bigcup_{\langle \mathtt{sync}, n_1, n_2\rangle \in \alpha} \{\mathtt{sync}\} \times \mu(n_1) \times \mu(n_2) \\
\pi[\mu] &= \textstyle\bigcup_{\langle\langle \mathtt{sync}, n_1, n_2\rangle, n\rangle \in \pi} (\{\mathtt{sync}\} \times \mu(n_1) \times \mu(n_2)) \times \mu(n) \cup \\
&\qquad \textstyle\bigcup_{\langle\langle n_1, \mathtt{f}, n_2\rangle, n\rangle \in \pi} (\mu(n_1) \times \{\mathtt{f}\} \times \mu(n_2)) \times \mu(n)
\end{aligned}$$

4.4 Interprocedural Analysis

The interprocedural analysis computes a transfer function for each method invocation statement. We assume a method invocation site of the form $\mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$, a potentially invoked method `op` with formal parameters $\mathtt{v}_0, \ldots, \mathtt{v}_k$ with corresponding parameter nodes $n_{\mathtt{v}_0}, n_{\mathtt{v}_1}, \ldots, n_{\mathtt{v}_k}$, a parallel interaction graph $\langle\langle O_1, I_1, e_1\rangle, \tau_1, \alpha_1, \pi_1\rangle$ at the program point before the method invocation site, and a graph $\langle\langle O_2, I_2, e_2\rangle, \tau_2, \alpha_2, \pi_2\rangle$ from the `exit` statement of `op`. The interprocedural analysis has two steps. It first computes a mapping μ for the outside nodes from the callee. It then uses μ to combine the two parallel interaction graphs to obtain the parallel interaction graph at the program point immediately after the method invocation. The analysis computes μ as the least fixed point of the following constraints:

[4]The only exception is in the definition of $O[\mu]$ where we do not substitute the load node n_L that constitutes the end point of an outside edge $\langle n, \mathtt{f}, n_L\rangle$.

$$I_1(l_i) \subseteq \mu(n_{\mathrm{v}_i}), \forall i \in \{0, 1, \ldots k\} \quad (1)$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in O_2, \langle n_3, \mathtt{f}, n_4\rangle \in I_1, n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)} \quad (2)$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in O_2, \langle n_3, \mathtt{f}, n_4\rangle \in I_2, \quad \mu(n_1) \cap \mu(n_3) \neq \emptyset, n_1 \neq n_3}{\mu(n_4) \cup \{n_4\} \subseteq \mu(n_2)} \quad (3)$$

The first constraint initializes μ; the next two constraints extend μ. Constraint 1 maps each parameter node from the callee to the nodes from the caller that represent the actual parameters at the call site. Constraint 2 matches outside edges read by the callee against corresponding inside edges from the caller. Constraint 3 matches outside edges from the callee against inside edges from the callee to model aliasing between callee nodes.

The algorithm next extends μ to μ' to ensure that all nodes from the callee (except the parameter nodes) appear in the new parallel interaction graph:

$$\mu'(n) = \begin{cases} \mu(n) & \text{if } n \in N_P \\ \mu(n) \cup \{n\} & \text{otherwise} \end{cases}$$

The algorithm computes the new parallel interaction graph $\langle\langle O', I', e'\rangle, \tau', \alpha', \pi'\rangle$ at the program point after the method invocation as follows:

$$O' = O_1 \cup O_2[\mu'] \qquad I' = I_1 \cup (I_2 - V \times N)[\mu']$$
$$\tau' = \tau_1 \cup \tau_2[\mu'] \qquad \alpha' = \alpha_1 \cup \alpha_2[\mu']$$
$$\pi' = \pi_1 \cup \pi_2[\mu'] \cup (O_2[\mu'] \cup \alpha_2[\mu']) \times \tau_1$$

It computes the new escape function e' as the union of the escape function e_1 before the method invocation and the expansion of the escape function e_2 from the callee through μ'. More formally, the following constraints define the new escape function e' as

$$e_1(n) \subseteq e'(n) \qquad \frac{n_2 \in \mu'(n_1)}{(e_2(n_1) - N_P)[\mu'] \subseteq e'(n_2)}$$

propagated over the edges from $O' \cup I'$. After the interprocedural analysis, reachability from the parameter nodes of the callee is no longer relevant for the escape function, hence the set difference in the second initialization constraint. We have a proof that this interprocedural analysis produces to a parallel interaction graph that is at least as conservative as the one that would be obtained by inlining the callee and performing the intraprocedural analysis as in section 4.2 [20].

Finally, we simplify the resulting parallel interaction graph by removing superfluous nodes and edges. We remove all load nodes n_L such that $e'(n_L) = \emptyset$ from the graph; such load nodes do not represent any concrete object. We also remove all all outside edges $\langle n_1, \mathtt{f}, n_2\rangle$ that start from a captured node n_1 (where $e'(n_1) = \emptyset$); such outside edges do not represent any concrete reference. Finally, we remove all nodes that are not reachable from a live variable, parameter node, or unanalyzed started thread node from τ'.

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

The analysis uses a worklist algorithm to solve the combined intraprocedural and interprocedural dataflow equations. A bottom-up analysis of the program yields the full result with one analysis per strongly connected component of the call graph. Within strongly connected components, the algorithm iterates to a fixed point.

4.5 Thread Interaction

Interactions between threads take 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 $\langle\langle O, I, e\rangle, \tau, \alpha, \pi\rangle$ from a program point in the starter thread, a node n_T that represents the startee thread, and a `run` method that runs when the thread object represented by n_T starts. The parallel interaction graph associated with the `exit` statement of the `run` method is $\langle\langle O_2, I_2, e_2\rangle, \tau_2, \alpha_2, \pi_2\rangle$. The result of the thread interaction algorithm is a parallel interaction graph $\langle\langle O', I', e'\rangle, \tau', \alpha', \pi'\rangle$ that models all the interactions between the execution of the starter thread (up to its corresponding program point) and the entire startee thread. This result conservatively models all possible interleavings of the two threads.

The algorithm has two steps. It first computes two mappings μ_1, μ_2, where μ_1 maps outside nodes from the starter and μ_2 maps outside nodes from the startee. It then uses μ_1 and μ_2 to combine the two parallel interaction into a single parallel interaction graph that reflects the interactions between the two threads. The algorithm computes μ_1 and μ_2 as the least fixed point of the following constraints:

$$n_T \in \mu_2(n_{\mathrm{v}_0}), n_T \in \mu_2(n_{\mathrm{CT}}) \quad (4)$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in O_i, \langle n_3, \mathtt{f}, n_4\rangle \in I_j, n_3 \in \mu_i(n_1)}{n_4 \in \mu_i(n_2)} \quad (5)$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in O_i, \langle n_3, \mathtt{f}, n_4\rangle \in I_i, \quad \mu_i(n_1) \cap \mu_i(n_3) \neq \emptyset, n_1 \neq n_3}{\mu_i(n_4) \cup \{n_4\} \subseteq \mu_i(n_2)} \quad (6)$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in I_i, \langle n_3, \mathtt{f}, n_4\rangle \in O_j, n_3 \in \mu_i(n_1)}{n_2 \in \mu_j(n_4)} \quad (7)$$

$$\frac{n_2 \in \mu_i(n_1), n_3 \in \mu_j(n_2)}{n_3 \in \mu_i(n_1)} \quad (8)$$

Here n_{v_0} is the parameter node associated with the single parameter of the `run` method – the `this` pointer – and n_{CT} is the dummy current thread node. Also, $I_1 = I$ and $O_1 = O \cap (\pi @ n_T)$. Note that the algorithm computes interactions only for outside edges from the starter thread that represent references read after the startee thread starts.

Unlike the caller/callee interaction, where the execution of the caller is suspended during the execution of the callee, in the starter/startee interaction, both threads execute in parallel, producing a more complicated set of statement interleavings. The interthread analysis must therefore model a richer set of potential interactions in which each thread can read edges created by the other thread. The interthread

analysis therefore uses two mappings (one for each thread) instead of just one mapping. It also augments the constraints to reflect the potential interactions.

In the same style as in the interprocedural analysis, the algorithm first initializes the mappings μ_1', μ_2' to extend μ_1 and μ_2, respectively. Each node from the two initial parallel interaction graphs (except n_{v_0}) will appear in the new parallel interaction graph:

$$\mu_1'(n) = \mu_1(n) \cup \{n\}$$
$$\mu_2'(n) = \begin{cases} \mu_2(n) & \text{if } n = n_{v_0} \\ \mu_2(n) \cup \{n\} & \text{otherwise} \end{cases}$$

The algorithm uses μ_1' and μ_2' to compute the resulting parallel interaction graph as follows:

$$O' = O[\mu_1'] \cup O_2[\mu_2'] \qquad I' = I[\mu_1'] \cup (I_2 - V \times N)[\mu_2']$$
$$\tau' = \tau[\mu_1'] \cup \tau_2[\mu_2'] \qquad \alpha' = \alpha[\mu_1'] \cup \alpha_2[\mu_2']$$
$$\pi' = \pi[\mu_1'] \cup \pi_2[\mu_2'] \cup (O_2[\mu_2'] \cup \alpha_2[\mu_2']) \times \tau[\mu_1'] \cup \pi@n_T[\mu_1'] \times \tau_2[\mu_2']$$

In addition to combining the action orderings from the starter and startee, the algorithm also updates the new action order π' to reflect the following ordering relationships:

- All actions and outside edges from the startee occur in parallel with all of the starter's threads, and
- All actions and outside edges from the starter thread that occur in parallel with the startee thread also occur in parallel with all of the threads that the startee starts.

The new escape function e' is the union of the escape function e from the starter and the escape function e_2 from the startee, expanded through μ_1 and μ_2, respectively. More formally, the escape function e' is initialized by the following two constraints

$$\frac{n_2 \in \mu_1(n_1)}{e(n_1)[\mu_1] \subseteq e'(n_2)} \qquad \frac{n_2 \in \mu_2(n_1)}{(e_2(n_1) - N_P)[\mu_2] \subseteq e'(n_2)}$$

and propagated over the edges from $O' \cup I'$.

4.6 Interthread Analysis

The interthread analysis uses a fixed-point algorithm to obtain a single parallel interaction graph that reflects the interactions between all of the parallel threads. The algorithm repeatedly chooses a node $n_T \in \tau$, retrieves the analysis result from the `exit` node of the corresponding `run` method,[5] then uses the thread interaction algorithm presented above in Section 4.5 to compute the interactions between the analyzed threads and the thread represented by n_T and combine the two parallel interaction graphs into a new graph. Once the algorithm reaches a fixed point, it removes all nodes in N_T from the escape function — the final graph already models all of the possible interactions that may affect nodes that escape only via unanalyzed thread nodes. The analysis may therefore recapture thread nodes that escaped before the interthread analysis. For example, if a thread node does not escape via a parameter node, it is captured after the interthread analysis. Finally the algorithm enhances the efficiency and precision of the analysis by removing superfluous nodes and edges using the same simplification method as in the interprocedural analysis.

As presented, the algorithm assumes that each node $n \in \tau$ represents multiple instances of the corresponding thread. Our implementation improves the precision of the analysis by tracking whether each node represents a single thread or multiple threads. For nodes that represent a single thread, the algorithm computes the interactions just once, adjusting the new action order π' to record that the outside edges and actions from the startee thread do not occur in parallel with the node n that represents the startee thread. For nodes that represent multiple threads, the algorithm repeatedly computes the interactions until it reaches a fixed point.

4.7 Resolving Outside Nodes

It is possible to augment the algorithm so that it records, for each outside node, all of the inside nodes that it represents during the analysis of the entire program. 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. In the absence of nodes that escape via unanalyzed threads or methods, this enables the algorithm to obtain complete, precise points-to information even for analysis results that contain outside nodes.

5. ANALYSIS USES

We next discuss how we use the analysis results to perform two optimizations: region reference check elimination and synchronization elimination.

5.1 Region Reference Check Elimination

The analysis eliminates region reference checks by verifying that no object allocated in a given region escapes the computation that executes in the context of that region. In our system, all such computations are invoked via the execution of a statement of the form `r.enter(t)`. This statement causes the the `run` method of the thread `t` to execute in the context of the memory region `r`. The analysis first locates all of these `run` methods. It then analyzes each `run` method, performing both the intrathread and interthread analysis, and checks that none of the inside nodes in the analysis result escape. If none of these inside nodes escape, all of the objects allocated inside the region are inaccessible when the computation terminates. All of the region reference checks will therefore succeed and can be removed.

5.2 Synchronization Elimination

The synchronization elimination algorithm uses the results of the interthread analysis to find captured objects whose synchronization operations can be removed. Like previous synchronization elimination algorithms, our algorithm uses the intrathread analysis results to remove synchronizations on objects that do not escape the thread that created them. Unlike previous synchronization elimination algorithms, our algorithm also analyzes the interactions between parallel threads. It then uses the action set α and the action ordering relation π to eliminate synchronizations on objects with synchronizations from multiple threads.

[5] The algorithm uses the type information to determine which class contains this `run` method. For inside nodes, this approach is exact. For outside nodes, the algorithm uses class hierarchy analysis to find a set of classes that may contain the `run` method. The algorithm computes the interactions with each of the possible `run` methods, then merges the results. In practice, τ almost always contains inside nodes only — the common coding practice is to create and start threads in the same method.

The analysis proceeds as follows. For each node n that is captured after the interthread analysis, it examines π to find all threads t that execute in parallel with a synchronization on n. It then examines the action set α to determine if t also synchronizes on n. If none of the parallel threads t synchronize on n, the compiler can remove all synchronizations on the objects that n represents. Even if multiple threads synchronize on these objects, the analysis has determined that the synchronizations are temporally separated by thread start events and therefore redundant.

6. EXPERIMENTAL RESULTS

We have implemented our combined pointer and escape analysis algorithm in the MIT Flex compiler system, a static compiler for Java. We used the analysis information for synchronization elimination and elimination of dynamic region reference checks. We present experimental results for a set of multithreaded benchmark programs. In general, these programs fall into two categories: web servers and scientific computations. The web servers include Http, an http server, and Quote, a stock quote server. Both of these applications were written by others and posted on the Internet. Our scientific programs include Barnes and Water, two complete scientific applications that have appeared in other benchmark sets, including the SPLASH-2 parallel computing benchmark set [23]. We also present results for two synthetic benchmarks, Tree and Array, that use object field assignment heavily. These benchmarks are designed to obtain the maximum possible benefit from region reference check elimination.

6.1 Methodology

We first modified the benchmark programs to use region-based allocation. The web servers create a new thread to service each new connection. The modified versions use a separate region for each connection. The scientific programs execute a sequence of interleaved serial and parallel phases. The modified versions use a separate region for each parallel phase. The result is that all of the modified benchmarks allocate long-lived shared objects in the garbage-collected heap and short-lived objects in regions. The modifications were relatively straightforward to perform, but it was difficult to evaluate the correctness of the modifications without the static analysis. The web servers were particularly problematic since they heavily use the Java libraries. Without the static analysis it was not clear to us that the libraries would work correctly with region-based allocation. For Http, Quote, Tree, and Array, the interprocedural analysis alone was able to verify the correct use of region-based allocation and enable the elimination of all dynamic region checks. Barnes and Water required the interthread analysis to eliminate the checks — interprocedural analysis alone was unable to verify the correct use of region-based allocation.

We used the MIT Flex compiler to generate a C implementation of each benchmark, then used gcc to compile the program to an x86 executable. We ran the Http and Quote servers on a 400 MHz Pentium II running Linux, with the clients running on an 866 MHz Pentium III running Linux. The two machines were connected with their own private 100 Mbit/sec Ethernet. We ran Water, Barnes, Tree, and Array on an 866 MHz Pentium III running Linux.

Program	Bytecode instructions	Analysis time [s] for removing checks	Analysis time [s] for removing syncs	Backend time [s]
Tree	10,970	0.5	15.9	41.1
Array	10,896	0.6	16.9	42.2
Water	17,675	11.3	56.1	66.0
Barnes	15,945	6.9	94.2	54.8
Http	14,313	17.1	38.3	73.8
Quote	14,039	16.9	41.4	61.4

Figure 10: Program Sizes and Analysis Times

Program	Original version	Optimized version	
		Interprocedural	Interthread
Tree	59	43	43
Array	59	43	43
Water	2,367,193	919,575	919,575
Barnes	2,838,720	678,355	678,355
Http	67,268	8,460	7,406
Quote	268,913	200,650	198,610

Figure 11: Number of Synchronization Operations

Program	Standard	Checks	No Checks
Tree	6.5	16.8	7.0
Array	8.2	43.4	8.3
Water	9.6	9.7	8.1
Barnes	8.4	7.6	6.7
Http	4.5	5.3	5.2
Quote	11.7	11.3	11.3

Figure 12: Execution Times for Benchmarks

Program	Number of Objects in Heap	Number of Objects in Regions
Tree	184	65,534
Array	183	8
Water	20,755	3,110,675
Barnes	17,622	2,121,167
Http	12,228	62,062
Quote	21,785	121,350

Figure 13: Allocation Statistics for Benchmarks

6.2 Results

Figure 10 presents the program sizes and analysis times. The synchronization elimination algorithm analyzes the entire program, while the region check algorithm analyzes only the `run` methods and the methods that they (transitively) invoke. The synchronization elimination analysis therefore takes significantly more time than the region analysis. The backend time is the time required to produce an executable once the analysis has finished. All times are in seconds. Figure 11 presents the number of synchronizations for the Original version with no analysis, the Interprocedural version with interprocedural analysis only, and the Interthread version with both interprocedural and interthread analysis. For this optimization, the interthread analysis produces almost no additional benefit over the interprocedural analysis. Figure 12 presents the execution times of the benchmarks. The Standard version allocates all objects in the garbage-collected heap and does not use region-based allocation. The Checks version uses region-based allocation with all of the dynamic checks. The No Checks version uses region-based

allocation with the analysis eliminating all dynamic checks. None of the versions uses the synchronization elimination optimization. Check elimination produces substantial performance improvements for Tree and Array and modest performance improvements for Water and Barnes. The running times of Http and Quote are dominated by thread creation and operating system overheads, so check elimination provides basically no performance increase. Figure 13 presents the number of objects allocated in the garbage-collected heap and the number allocated in regions. The vast majority of the objects are allocated in regions.

6.3 Discussion

Our applications use regions in one of two ways. The servers allocate a new region for each connection. The region holds the new objects required to service the connection. Examples of such objects include `String` objects that hold responses sent to clients and iterator objects used to find requested data. The scientific programs use regions for auxiliary objects that structure the parallel computation. These objects include the `Thread` objects required to generate the parallel computation and objects that hold values produced by intermediate calculations.

In general, eliminating region checks provides modest performance improvements. We therefore view the primary value of the analysis in this context as helping the programmer to use regions correctly. We expect the analysis to be especially useful in situations (such as our web servers) when the programmer may not have complete confidence in his or her detailed knowledge of the program's object usage patterns.

7. RELATED WORK

We discuss several areas of related work: analysis of multithreaded programs, escape analysis for multithreaded programs, and region-based allocation.

7.1 Analysis of Multithreaded Programs

The analysis of multithreaded programs is a relatively unexplored field [17]. There is an awareness that multithreading significantly complicates program analysis 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 [10], and Knoop, Steffen and Vollmer present an efficient dataflow analysis framework for bit-vector problems such as liveness, reachability and available expressions [12]. Both frameworks are designed for programs with structured, parbegin/parend concurrency and are intraprocedural. We view the main contributions of this paper as largely orthogonal to this previous research. In particular, the main contributions of this paper center on abstractions and algorithms for the interprocedural and compositional analysis of programs with unstructured multithreading. We also focus on problems, pointer and escape analysis, that do not fit within either framework.

We are aware of two pointer analysis algorithms for multithreaded programs: an algorithm by Rugina and Rinard for multithreaded programs with structured parbegin/parend concurrency [19], and an intraprocedural algorithm by Corbett [7]. The algorithms are not compositional (they discover the interactions between threads by repeatedly reanalyzing each thread in each new analysis context to reach a fixed point), do not maintain escape information, and do not support the analysis of incomplete programs.

7.2 Escape Analysis for Multithreaded Programs

Published escape analysis algorithms for Java programs do not analyze interactions between threads [3, 6, 22, 2]. If an object escapes via a thread object, it is never recaptured. These algorithms are therefore best viewed as sequential program analyses that have been extended to execute correctly but very conservatively in the presence of multithreading. Our analysis takes the next step of analyzing interactions between threads to recapture objects accessed by multiple threads.

Ruf's analysis occupies a point between traditional escape analyses and our multithreaded analysis [18]. His analysis tracks the synchronizations that each thread performs on each object, enabling the compiler to remove synchronizations for objects accessed by multiple threads if only one thread synchronizes on the object. Our analysis goes a step further to remove synchronizations even if multiple threads synchronize on the object. The requirement is that thread start events must temporally separate synchronizations from different threads.

7.3 Region-Based Allocation

Region-based allocation has been used in systems for many years. Our comparison focuses on safe versions, which ensure that there are no dangling references to deleted regions. Several researchers have developed type-based systems that support safe region-based allocation [21, 8]. These systems use a flow-insensitive, context-sensitive analysis to correlate the lifetimes of objects with the lifetimes of computations. Although these analyses were designed for sequential programs, it should be straightforward to generalize them to handle multithreaded programs.

Gay and Aiken's system provides an interesting contrast to ours in its overall approach [9]. They provide a safe, flat region-based system that allows arbitrary references between regions. The implementation instruments each store instruction to count references that go between regions. A region can be deleted only when there are no references to its objects from objects in other regions. This dynamic, reference counted approach works equally well for both sequential and multithreaded programs. The system also supports the explicit assignment of objects to regions and allows the programmer to use type annotations to specify that a given reference must stay within the same region. Violations of this constraint generate a run-time error; a static analysis reduces but is not designed to eliminate the possibility of such an error occurring.

Following the Real-Time Java specification, our implementation provides a less flexible system of hierarchically organized regions with an implicit assignment of objects to regions. Because region lifetimes are hierarchically nested, the implementation dynamically counts, for each region, the number of child regions rather than the number of external pointers into each region. Instead of performing counter manipulations at each store, the unoptimized version of our system checks each assignment to ensure that the program never generates a reference that goes down the hierarchy from an ancestor region to a descendant region. Our static analysis eliminates these checks, with the interthread analysis required to successfully optimize multithreaded programs.

8. CONCLUSION

Multithreading is a key program structuring technique, language and system designers have made threads a central part of widely used languages and systems, and multithreaded software is becoming pervasive. This paper presents an abstraction (parallel interaction graphs) and an algorithm that uses this abstraction to extract precise points-to, escape, and action ordering information for programs that use the standard unstructured form of multithreading provided by modern languages and systems. We have implemented the analysis in the MIT Flex compiler for Java, and used the extracted information to verify that programs correctly use region-based allocation constructs, eliminate dynamic checks associated with the use of regions, and eliminate unnecessary synchronization. Our experimental results show that analyzing the interactions between threads significantly increases the effectiveness of the optimizations for region-based programs, but has little effect for synchronization elimination.

9. ACKNOWLEDGEMENTS

We started this research in collaboration with John Whaley; we thank him for his contributions during this collaboration. We thank Wes Beebee for his invaluable assistance with the region-based allocation package and Darko Marinov and Viktor Kuncak for many interesting discussions regarding pointer and escape analysis.

10. REFERENCES

[1] O. Agesen, J. Palsberg, and M. Schwartzbach. Type inference of SELF: analysis of objects with dynamic and multiple inheritance. *Software—Practice and Experience*, 25(9):975–995, Sept. 1995.

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

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

[4] G. Bollella, B. Brosgol, S. Furr, D. Hardin, P. Dibble, J. Gosling, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, Mass., 2000.

[5] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

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

[7] 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*, Mar. 1998.

[8] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, Jan. 1999.

[9] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[10] 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.

[11] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[12] 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.

[13] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Mass., 2000.

[14] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[15] V. Pai, P. Druschel, and W. Zwaenepol. Flash: An efficient and portable web server. In *Proceedings of the Usenix 1999 Annual Technical Conference*, June 1999.

[16] J. Reppy. *Higher–order Concurrency*. PhD thesis, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., June 1992.

[17] M. Rinard. Analysis of multithreaded programs. In *Proceedings of 8th Static Analysis Symposium*, Paris, France, July 2001.

[18] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[19] 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.

[20] A. Sălcianu. Pointer analysis and its applications for Java programs. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, In preparation.

[21] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4), July 1998.

[22] 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, Nov. 1999.

[23] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*. ACM, New York, June 1995.