Region Analysis and Transformation for Java Programs

Sigmund Cherem and Radu Rugina
Computer Science Department
Cornell University
Ithaca, NY 14853
{siggi,rugina}@cs.cornell.edu

ABSTRACT

This paper presents a region analysis and transformation framework for Java programs. Given an input Java program, the compiler automatically translates it into an equivalent output program with region-based memory management. The generated program contains statements for creating regions, allocating objects in regions, removing regions, and passing regions as parameters. As a particular case, the analysis can enable the allocation of objects on the stack.

Our algorithm uses a flow-insensitive and context-sensitive points-to analysis to partition the memory of the program into regions and to identify points-to relations between regions. It then performs a flow-sensitive, inter-procedural region liveness analysis to identify object lifetimes. Finally, it uses the computed region information to produce the region annotations in the output program. Our results indicate that, for several of our benchmarks, the transformation can allocate most of the data on stack or in short-lived regions, and can yield substantial memory savings.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Memory management; F.3.2 [**Semantics of Prog. Languages**]: Program Analysis

General Terms

Languages, Performance

Keywords

Region-based memory management, pointer analysis, program transformations.

1. INTRODUCTION

A region-based system groups heap objects together in regions and deallocates all of the objects in a region at once. This approach has a number of appealing properties. First, it may improve data locality. Second, it may improve performance by deallocating entire regions rather than individual objects. Third, it can be used in conjunction with static analysis to provide automatic memory management. More precisely, the compiler can group heap data into regions, it can statically determine the program points where it is safe to reclaim regions, and then transform the program by augmenting it with region annotations. This approach, originally proposed by Tofte and Talpin [23, 2], has been successfully applied to functional languages such as ML.

And last, but not least, region-based memory management is an attractive approach to memory management for real-time systems, since such systems cannot afford to be interrupted for unbounded amounts of time by a garbage collector. For instance, the Real-Time Specification for Java (RTSJ) [5] allows real-time programs to manage data without garbage collection by allocating objects into an immortal memory area, or into scoped regions of memory with bounded lifetimes. However, RTSJ requires run-time checks to ensure the safe access and deallocation of objects in such memory areas. A static region analysis has appealing properties in this context: it can automatically infer memory scopes, freeing the programmer of the burden of reasoning about object lifetimes; or, it can enable the optimization of programs with explicit scopes by eliminating unnecessary run-time checks.

This paper presents a region analysis and transformation system for Java programs: it proposes techniques for handling the imperative, object-oriented, and multithreaded constructs in the language. Given an input Java program, our system produces an output program augmented with region annotations. These annotations include creating and removing regions [1], allocating objects into regions, and passing regions as parameters.

Our region-based system has two features that distinguish it from several existing approaches: a) regions are not lexically scoped (unlike [23, 10, 15, 6, 8]); and b) the system allows dangling references, so regions are being reclaimed when they are no longer needed, even if there are incoming references from live regions (unlike [15, 6, 8, 5]). Both choices improve the precision of the computed lifetimes, at the expense of making the analysis more complex. Analysis systems with these features [1, 16] have been studied in a functional setting, but not for imperative languages.

As in the case of many other analysis problems, the main complication that imperative languages bring over functional ones is the presence of pointer-based structures, aliasing,

[1]We use the terms *region creation* and *removal*, instead of *allocation* and *deallocation*, in order to avoid ambiguities between object allocation and region allocation.

and destructive updates. We leverage *pointer analysis* techniques to address all of these issues. We use a unification-based flow-insensitive, but context-sensitive pointer analysis to partition the memory into regions and compute points-to relations between the regions. Further, in the presence of imperative control-flow constructs such as loops, sequences, or if statements, the compiler must use a flow analysis to identify live regions at each program point. We propose a flow-sensitive, inter-procedural *region liveness analysis* that computes region lifetimes. The compiler uses this information to determine the program points where to insert region creation and removal statements.

Finally, object-oriented and other features of Java pose further challenges to the analysis. Virtual methods may dynamically invoke different methods at run-time; threads make it difficult to identify the lifetimes of shared objects; and exceptions complicate the control flow. This paper proposes a set of analysis techniques and run-time mechanisms to handle all of these constructs.

Region analysis provides a solution to the problem of stack allocation, as a particular case: it is able to identify regions whose lifetimes match methods lifetimes and can allocate these objects on stack. The analysis, however, does not focus exclusively on such situations; in general, it manages regions with arbitrary lifetimes, such as regions whose lifetimes span across multiple procedures, or regions with loop-carried lifetimes. Existing approaches to stack allocation [3, 9, 25, 4, 14, 21] are not aimed at identifying or taking advantage of these general cases.

We have implemented and evaluated our analysis and transformation on a set of Java benchmarks. We have extended the interpreter of the Kaffe VM [26] to support region annotations and to provide region run-time support. Our results indicate that, for almost all of the benchmarks where data is not live throughout the program, the transformation is able to allocate most of it on stack or in short-lived regions, and can yield substantial memory savings.

The paper is structured as follows. Section 2 presents our region constructs, Section 3 discusses an example, Section 5 presents the basic analysis algorithm, and Section 6 describes extensions. Finally, we present experimental results in Section 7, discuss related work in Section 8 and conclude in Section 9.

2. REGION CONSTRUCTS

The input language to our analysis and transformation system is Java. Given a input Java program, the system produces an output program which extends the original program with region annotations. In this section we present the region constructs informally. The formal semantics for similar constructs in a generic imperative language can be found in [20]. There are five region constructs:

- `create r` and `remove r`, where `r` is a variable that holds a handle to a region. Statement `create r` dynamically creates a new region and stores its handle in `r`; statement `remove r` dynamically removes region `r` and reclaims all of its memory. Region variables need not be declared; their scope is the enclosing method. We write `create r1,..,rn` as a shorthand for `create r1;..;create rn` (similarly for `remove`).
- `new C(e1,..,en) in r`, for the allocation of objects into regions. Here `C` is a class name, `e1`,.., `en` are the constructor actual parameters, and `r` is the region where the newly created object is to be placed. All allocation statements must specify a target region.
- `T m<r1,..,rk>(T1 p1,..,Tn pn) {..}`, for method declarations. Here `m` is the declared method; `r1`, .., `rk` are *region parameters* (in angle brackets); `T`, `T1`, .., `Tn` are standard Java types; and `p1`, ..., `pn` are the standard method parameters. We require overridden methods to have the same number of region parameters in the subclasses (because they may be dynamically dispatched). Intuitively, region parameters hold handles to regions where the method uses to place new objects into. We refer to all of the regions that are not parameter regions, but occur in the method body, as *local regions.*
- `m <r1,..,rk> (e1,..,en)`, for method calls. Here, `m` is the method, `r1`, .., `rn` are the actual region parameters, and `e1`, ..., `en` are the standard actual parameters of the method.

The execution of the program generates an error at run-time if: 1) a region `r` which occurs in a `remove` or `new` statement, or in a method call doesn't hold a handle to an allocated region (either because the region has not been created yet, or because it has already been removed); or 2) the program accesses a field or a method of an object which has been placed in a region that has been removed since the allocation of the object; or 3) the program executes a `create r` statement, but `r` already holds a handle to an allocated region. The analysis and translation in our system ensures that none of these situations will occur when the output program is being executed.

3. EXAMPLE

This section presents two examples that illustrate the main features of our analysis and transformation.

3.1 Example 1: A List Container

Figure 1 presents two classes that implement a linked list structure. The code shown is the result of the transformation; the input program is the same, but without the region annotations. Class `Element` implements the list elements, which hold references to data objects. Class `List` holds the list head and provides three methods: `add`, to insert an element at the beginning of the list; `reverse`, to produce a new list with the elements in reverse order; and `iterator`, which creates an iterator over the list elements. We omit the definition of the `Iterator` class.

Our transformation generates the following region annotations for class `List`: it produces formal region parameters for methods `add`, `reverse` and `iterator` (at lines 11, 14, and 24); it annotates the object allocation sites (at lines 12, 15, and 25) with regions for the new objects; and passes a region parameter (at line 18) when method `reverse` invokes `add`.

For method `add`, region parameter `r1` represents the region where the method allocates the new list element that will hold the data object `e`. Method `reverse` requires two region parameters: region `r2` holds the new list object, and region `r3` holds all the elements of the new list. The objects that the method creates in these two regions are being returned to the caller. The reversal method allocates the list object in `r2` at

```
 1  class Element {
 2     Object data;
 3     Element next;
 4     public Element(Object d, Element n)
 5       { data = d; next = n; }
 6  }
 7
 8  class List {
 9     Element head;
10
11     public void add<r1>(Object e)
12       { head = new Element(e,head) in r1; }
13
14     public List reverse<r2,r3>(){
15        List list = new List() in r2;
16        Element elem = head;
17        while (elem != null) {
18           list.add<r3>(elem.data);
19           elem = elem.next;
20        }
21        return list;
22     }
23
24     public Iterator iterator<r4>()
25       { return new Iterator(head) in r4; }
26  }
```

Figure 1: Example translation. The input program is the same, but without the region annotations.

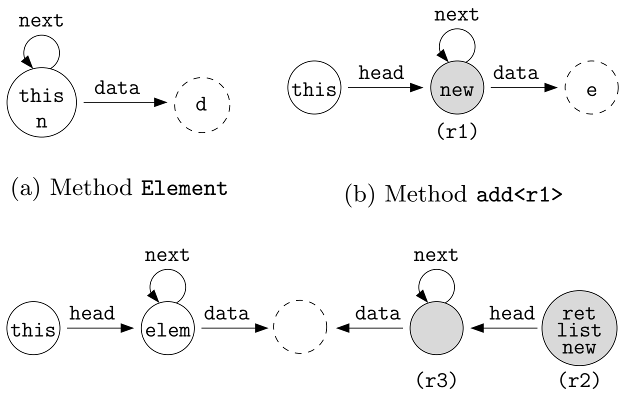
line 15, but delegates method `add` to perform the allocation of list elements at line 12; therefore, it passes region `r3` as an actual parameter to `add`. Finally, method `iterator` has one region parameter `r4`; it creates a new container in this region and returns it to its caller.

The transformation generates no region annotations for class `Element`. Nonetheless, the compiler must analyze its constructor method to figure out how its execution affects the aliasing and region liveness in its callers.

3.2 Analysis Information

The key piece of information that the analysis computes is a *region points-to graph* for each method. This graph concisely captures the *effects* of invoking the method; these include points-to effects, object allocation effects, and access (i.e., read and write) effects for the method. The nodes in a region points-to graph represent the regions that the execution of the method (including all invoked methods) either accesses or manipulates references into. The edges model points-to relations between these regions. Edge labels represent field names, and node labels represent sets of variables (or allocation sites) that reference (or create) objects in the corresponding region. For this example, we denote allocation sites using `new`, since there is at most one allocation per method. The special variable `this` models the receiver object; and `ret` models the return value. Nodes with no labels represent regions that the method indirectly accesses through calls to other methods.

Figure 2 shows the points-to graphs that our analysis computes for the constructor of class `Element`, and for methods `add` and `reverse` in class `List`. Shaded nodes in these graphs indicate regions that the method – including all the methods it invokes – allocates new objects into (the *allocation effects*); and nodes drawn with dashed circles indicate regions that the method does not access (hence, solid circles describe *access effects*).



(a) Method `Element` (b) Method `add<r1>`

(c) Method `reverse<r2,r3>`

Figure 2: Points-to graphs for methods `Element`, `add`, and `reverse`.

Consider the graph for method `reverse` shown in Figure 2(c). In this graph, nodes `this` and `elem` represent the regions where the receiver object and its list elements are located when the method is invoked. The two shaded nodes on the right represent the regions where the method allocates the new list and its elements; the method must receive these regions (`r2` and `r3` in the code) from its caller. The dashed node in the middle shows that the original list and the reversed list share data items in the same region. The node representing region `r3` and its incoming and outgoing edges are generated by the call to `add`. Indeed, the right subgraph of reverse matches the points-to graph of method `add`. Finally, the pseudo-variable `ret` models the return value of this method and shows that the method returns a reference to the newly created list in `r2`.

Region points-to graphs provide the key information that the compiler needs to generate the output program. First, to determine the region for an allocation site, the compiler looks up the graph for the node corresponding to that site; for instance, the middle node in the graph of `add` generates the annotation "`in r1`" at line 12. Second, to determine how many and what region parameters each method needs, the compiler looks for all of the allocation (shaded) nodes reachable from the method parameters and the return value; for `reverse`, regions `r2` and `r3` are the two shaded nodes reachable from `this` and `ret`. Third, at call sites, the analysis matches up nodes representing actual values in the caller graph to nodes representing the corresponding formals in the callee graph, and uses this mapping to derive the actual regions needed at the call site. Finally, the analysis uses the access effects in these graphs to compute region liveness information in the callers of these methods, and place region creation and removal commands in those methods.

3.3 Example 2: Polynomials

Figure 3 shows an implementation of polynomials using complex coefficients. Class `Complex` implements complex numbers: it provides a constructor, and methods `mul` and `plus` to add and multiply complex numbers. Such numbers are immutable, so these operations return their results as new objects. Hence, each operation requires one region parameter to store the result.

```
27 class Polynomial {
28    List coeffs;
29    public void valueAt(Complex x) {
30      create r7,r8;
31      List rev = coeffs.reverse<r7,r8>();
32
33      create r9;
34      Complex sum = new Complex(0,0) in r9;
35
36      create r10;
37      Iterator it = rev.iterator<r10>();
38      while (!it.empty()) {
39        Complex coeff = (Complex)it.next();
40        create r11;
41        Complex tmp = sum.mul<r11>(x);
42        remove r9;
43        create r9;
44        sum = tmp.plus<r9>(coeff);
45        remove r11;
46      }
47      remove r7,r8,r10;
48      System.out.println(sum.re);
49      System.out.println(sum.im);
50      remove r9;
51    }
52 }
53
54 class Complex {
55    double re, im;
56    public Complex(double r, double i) {
57      re = r; im = i;
58    }
59    public Complex mul<r12>(Complex c) {
60       double r = re*c.re - im*c.im;
61       double i = re*c.im + im*c.re;
62       return new Complex(r,i) in r12;
63    }
64    public Complex plus<r13>(Complex c) {
65       double r = re+c.re, i = im*c.im;
66       return new Complex(r,c) in r13;
67    }
68 }
```

Figure 3: Polynomials with complex coefficients.

Class `Polynomial` implements a polynomial as a list of complex number coefficients, stored in ascending order. The method `valueAt` computes the value of the polynomial for a given value of its variable and prints the result out. For this, the method first reverses the coefficient list, then computes the result according to the formula:

$$\sum_{i=0}^{n} a_i x^i = (((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \ldots) \cdot x + a_0$$

The translation of this method illustrates several features of our analysis. The method receives no region parameters, but uses five local regions (`r7`, ..., `r11`) created and removed at different points in the method. Regions `r7` and `r8` hold the reversed list and its elements; region `r10` holds the iterator. The program removes all of these regions immediately after the loop.

Most interesting are regions `r11` and `r9`. The analysis identifies that `r11` holds a temporary result at each iteration. Therefore, it creates the region at the beginning of the loop body, at line 40, and removes it at the end of the loop body, at line 45. The region lifetime is subsumed by the lifetime of each loop iteration.

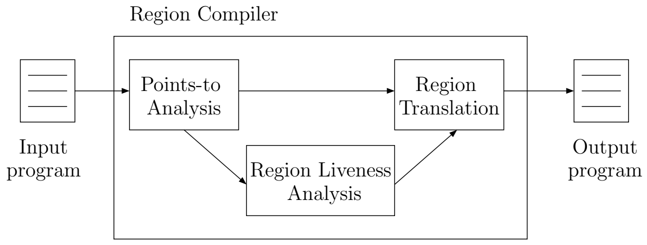


Figure 4: Region compilation system

Furthermore, it determines that region `r9` holds an object with a *loop-carried lifetime*: the value produced at the end of one iteration is needed at the beginning of the next iteration. It therefore generates a region removal statement first, at line 42, followed by a region creation statement, at next line, to store the new value. Intuitively, this resets the contents of the region. The translation also accounts for the start and the completion of this loop-carried dependence: in the first iteration, the region has already been created at line 33; after the last iteration, the program removes the region at line 50. Therefore, region `r9` is being created at two different points and removed at two other points in the program. However, the transformation ensures that the execution through any paths in the program alternates the region creation and removal; and that the program accesses data in this region only when the region is live.

The approaches based on lexically scoped regions will fail to accurately characterize the lifetimes of objects in region `r9`. They will be forced to place all of these objects into a long-lived region (which subsumes the lifetime of the whole loop), although only one single instance is live at any given point in time. Hence, regions with lexical scopes are inherently less flexible and less precise than our approach, which lexically decouples region creation from region removal.

4. SYSTEM OVERVIEW

Figure 4 presents an overview of our system. The overall system includes the following three main phases:

- *Phase 1: Points-to analysis.* In this phase, the compiler performs a flow-insensitive and context-sensitive analysis that partitions the memory into regions. For each method, it builds a region points-to graph that captures the effects of executing the method and all the methods it transitively invokes. The analysis result of each method is region-polymorphic and can be instantiated at each call site.

- *Phase 2: Region liveness analysis.* Next, the compiler performs a flow-sensitive analysis and computes live regions at each program point. The analysis simultaneously keeps track of live variables and live regions, and uses the computed points-to graphs to determine the reachability of regions from live variables. We present a basic analysis, as well an extension that enables inter-procedural region removal.

- *Phase 3: Region translation.* Finally, the compiler uses the points-to and the region liveness information computed in the previous phases to produce region annotations in the output program. The translation requires a single pass through the program.

5. ALGORITHM

This section presents each of the three phases in our system in detail. To simplify the presentation, we model each method in the program using a control-flow graph whose nodes are one of the following statements:

$$\begin{array}{lll} x = y & x = y.f & x.f = y \\ x = \texttt{new}\ C & \texttt{return}\ x & x = y_1.m(y_2, ..., y_n) \end{array}$$

where x, y, $y_1, \ldots, y_n$ are all program variables, f is a field name, and C is a class name. Variables include the implicit argument `this`. We model array accesses as field accesses on the special field: $f =$ `[]`.

5.1 Phase 1: Points-to Analysis

This phase computes a region points-to graph for each method. The nodes represent regions, and the edges represent points-to relations, as shown in the examples from Figure 2. Although the algorithm computes the points-to graphs and the allocation and access effects simultaneously, we present them separately for the sake of clarity: first we describe the pointer analysis alone, and then we discuss how it can be augmented to compute effects as well.

5.1.1 Intra-procedural Analysis

At the intra-procedural level, we use a standard flow-insensitive pointer analysis algorithm with unification constrains, due to Steensgaard [22]. We describe this algorithm in a formal setting and use this formalism to present the inter-procedural analysis in the next section.

The analysis of each method in the program works as follows. Let V be the set of variables of non-primitive types, S the set of allocation sites in the method, F the set of class fields, and P the set of statements in the method. We assume that V includes the implicit parameters `this`, as well as a pseudo-variable `ret` that models the return value of the method (if that is a reference). Let L be the set of node labels, consisting of allocation sites $a \in S$, variables $x \in V$, as well as field accesses $x.f \in V \times F$, that is, $L = S \cup V \cup (V \times F)$. Field accesses $x.f$ are relevant only when constructing the graphs, but not for performing the translation, which is why we omitted them in the examples from Section 3. Let L_P be the set of labels that occur in the method body.

Definition 1. *A points-to graph is $G = (N, E)$, where:*

- *$N \subseteq \mathcal{P}(L_P)$ is the set of region nodes such that each label occurs in at most one node. If $u \in L$ is a label, $n(u)$ is the graph node that contains u.*
- *$E = N \times F \times N$ is a set of edges labeled with field names. We write $\langle n, f, m\rangle$ to denote an edge between n and m on field f.*

Definition 2. *A consistent unification points-to graph is a points-to graph $G = (N, E)$ such that:*

(C1) $n(x.f) \in N \Rightarrow \langle n(x), f, n(x.f)\rangle \in E$, i.e., field access node labels are consistent with field labels on edges (consistency condition).

(C2) $\langle n, f, m\rangle, \langle n, f, m'\rangle \in G \Rightarrow m = m'$, i.e. a node can have at most one successor node on each field (unification condition).

$$\frac{\mathit{unify}(n, n') \quad m \neq m' \quad \langle n, f, m\rangle, \langle n', f, m'\rangle \in E}{\mathit{unify}(m, m')} \quad (R1)$$

$$\frac{\mathit{edge}(n, f, m) \quad \langle n, f, m'\rangle \in E \quad m \neq m'}{\mathit{unify}(m, m')} \quad (R2)$$

Figure 5: Rules for the intra-procedural analysis.

With these definitions, the goal of a unification-based pointer analysis is to build a consistent unification points-to graph such that, the right-hand side and the left-hand side of each assignment in the program correspond to the same node in the graph. We describe the algorithm that finds such a solution using the following two kinds of operations:

- *unify* (n, m), which unifies two nodes n and m in the graph. Given a graph $G = (N, E)$, it produces a new graph $G' = (N', E')$ such that:

$$\begin{array}{lcl} N' & = & N - \{n, m\} \cup \{n \cup m\} \\ E' & = & \{\langle p', f, q'\rangle \mid \exists \langle p, f, q\rangle \in E \ . \ p \subseteq p' \wedge q \subseteq q'\} \end{array}$$

- *edge* (n, f, m), which adds an edge between nodes n and m on field f. Given $G = (N, E)$, it produces a new graph $G' = (N, E \cup \langle n, f, m\rangle)$.

The algorithm proceeds as follows. First, it performs the following operations:

$$\begin{array}{ll} \mathit{edge}(n(x), f, n(x.f)) & \forall (x.f) \in L_P \\ \mathit{unify}(n(u), n(v)) & \forall (u = v) \in P \\ \mathit{unify}(n(\texttt{ret}), n(x)) & \forall (\texttt{return}\ x) \in P \end{array}$$

Next, the algorithm uses rules (R1) and (R2) from Figure 5 to propagate unifications down in the graph. When the algorithm performs the operation in the premise of a rule and the other premises hold, then it performs the operation(s) in the conclusion. Together, the two rules enforce the unification condition (C2) on the final points-to graph. This description naturally leads to an implementation which processes unification and edge operations using a worklist algorithm.

A key aspect is that the algorithm analyzes each method independently of its callers and produces an analysis result parameterized on the region parameters, which are assumed to be unaliased. This approach yields a general result, which can be instantiated at each call site by the inter-procedural analysis.

5.1.2 Inter-procedural Analysis

Once the compiler has built an intra-procedural points-to graph for each method, it analyzes call statements. It uses a context-sensitive analysis that embeds the information from the caller into the callee at each call site, according to the aliasing context at that site. We assume that the invoked method is statically known; Section 5.1.4 discusses how to handle virtual method calls.

We formulate the problem of analyzing a call site as follows. Given the a caller graph $G_r = (N_r, E_r)$ and a callee graph $G_e = (N_e, E_e)$, the analysis must construct a result graph $G'_r = (N'_r, E'_r)$ for the caller such that: a) G'_r is a conservative approximation of G_r; and b) the subgraph of G'_r

$$\frac{n(p_i) = n(p_j) \quad n(y_i) \neq n(y_j)}{\mathit{unify}\,(n(y_i), n(y_j))} \; (R3)$$

$$\frac{n(\mathtt{this}) = n(p_j) \quad n(y_0) \neq n(y_j)}{\mathit{unify}\,(n(y_0), n(y_j))} \; (R4)$$

$$\frac{n(\mathtt{ret}) = n(p_j) \quad n(x) \neq n(y_j)}{\mathit{unify}\,(n(x), n(y_j))} \; (R5)$$

$$\frac{\begin{array}{cc} \langle n_e, f, m_e \rangle \in E_e & \alpha(n_e) = n_r \\ \langle n_r, f, m_r' \rangle \in E_r & \alpha(m_e) = m_r \neq m_r' \end{array}}{\mathit{unify}\,(m_r, m_r')} \; (R6)$$

$$\frac{\begin{array}{cc} \langle n_e, f, m_e \rangle \in E_e & \alpha(n_e) = n_r \\ \langle n_r, f, m_r \rangle \in E_r & \alpha(m_e) \text{ undefined} \end{array}}{\alpha(m_e) \leftarrow m_r} \; (R7)$$

$$\frac{\begin{array}{cc} \langle n_e, f, m_e \rangle \in E_e & \alpha(n_e) = n_r \\ \forall p \,.\, \langle n_r, f, p \rangle \notin E_r & \alpha(m_e) = m_r \end{array}}{\mathit{edge}\,(n_r, f, m_r)} \; (R8)$$

$$\frac{\begin{array}{cc} \langle n_e, f, m_e \rangle \in E_e & \alpha(n_e) = n_r \\ \forall p \,.\, \langle n_r, f, p \rangle \notin E_r & \alpha(m_e) \text{ undefined} \\ \multicolumn{2}{c}{m = \mathit{FreshNode}(G_r)} \end{array}}{\alpha(m_e) \leftarrow m \quad \mathit{edge}\,(n_r, f, m)} \; (R9)$$

Figure 6: Rules for inter-procedural analysis.

rooted at the actual parameters is a conservative approximation of the subgraph of G_e rooted at the formal parameters of the callee. The resulting graph G_r' incorporates the effects of the call. Here we use the following definition.

DEFINITION 3. *A graph $G = (N, E)$ is a conservative approximation of $G' = (N', E')$ if there exists a node mapping $\alpha : N' \to N$ such that $\langle n, f, m \rangle \in E'$ implies $\langle \alpha(n), f, \alpha(m) \rangle \in E$ for all $n, m \in N'$.*

Consider a call statement $x = y_0.m(y_1, ..., y_n)$ that invokes a method `m` with formal parameters `this`, $p_1, \ldots, p_n$ and return value `ret`. To compute the result graph G_r', the analysis gradually builds a partial map $\alpha : N_e \to N_r$ that maps callee nodes to caller nodes. At the end, the mapping witnesses the embedding of the callee information into the caller.

The algorithm proceeds as follows. Initially, α maps each formal parameter of the callee to its corresponding actual parameter at the call site:

$$\begin{array}{rcll} \alpha(n(p_i)) & = & n(y_i) & \forall i = 1..n \\ \alpha(n(\mathtt{this})) & = & n(y_0) & \\ \alpha(n(\mathtt{ret})) & = & n(x) & \end{array}$$

To ensure that α is a map after this initialization, the analysis unifies actual nodes that correspond to parameter nodes that have been unified in the callee. For this, the algorithm uses rules (R3), (R4), and (R5) in Figure 6. These rules require that each parameter node has no more than one corresponding actual node in the caller; hence, they can be regarded as duals of rules (R1) and (R2), which require that each node has at most one successor for each given field.

After setting the initial values of α, the algorithm traverses the nodes in the callee graph G_e starting from the formal parameters and inspects each edge exactly once. During the traversal, it uses the rules (R6) – (R9) to complete the mapping and to trigger new operations. The algorithm traverses callee edges in a top-down fashion, such that the source node n_e of the callee is always mapped to a node n_r in the caller when the edge $\langle n_e, f, m_e \rangle$ is traversed. The four cases check if the edge target is mapped and if the mapping of the source already has an edge in the caller. In the last rule, when none holds, the analysis generates a fresh node in the caller.

The above algorithm shows how to analyze each call. The complete inter-procedural analysis uses a fixed-point approach to compute the final result. The compiler constructs the strongly connected components in the call graph and analyzes these components in reverse topological order. It iteratively analyzes the calls in each strongly connected component until no changes occur. Each strongly connected component in the call graph is analyzed exactly once.

5.1.3 Recursive Structures

For recursive methods that manipulate recursive data structures, rule (R9) in the interprocedural analysis may generate an unbounded number of fresh nodes, so the algorithm is no longer guaranteed to terminate. For example, consider a recursive method that traverses a list by following the next element at each recursive invocation:

```
static void traverse(Element h) {
   Element x = h.next;
   if (x != null) traverse(x);
}
```

As described so far, the inter-procedural analysis will keep re-analyzing method `traverse`, generating a fresh node at each iteration. Therefore, the analysis will loop forever.

Our solution to this problem is to maintain a Java type for each node in the graph and restrict the creation of new edges and nodes as follows. Whenever the analysis adds a new edge to the graph, it ensures that the target of that edge is not reachable from another node with the same type. Otherwise, it unifies the two nodes with the same type according to the following rule:

$$\frac{\begin{array}{cc} \mathit{edge}\,(n, f, m) & \langle m', m \rangle \in E^* \\ m' \neq m & \mathit{type}(m') = \mathit{type}(m) \end{array}}{\mathit{unify}\,(m', m)} \; (R10)$$

where E^* is the reflexive and transitive closure of the edge relation: $\langle m', m \rangle \in E^*$ means that m is reachable from m' in the points-to graph (which includes the new edge from n to m). The analysis implements rule (R10) by traversing the graph backwards from node n and looking for a node m' with the same type as m. This rule limits the number of graph nodes that model elements of recursive structures.

For instance, for method `traverse`, this rule unifies nodes `h` and `h.next`, because they have the same type `Element`. Therefore, the points-to graph for `traverse` will have a single node with a self-edge on field `next`. A similar situation occurs for the example from Section 3, where the analysis of the class constructor `Element` unifies nodes `this` and `n`.

The type associated with each region represents an upper bound for the actual types of object allocated in that region. The analysis keeps track of region types as follows. It initializes the types of variable nodes $n(x)$ and field nodes $n(x.f)$ to their declared types, and the types of allocation nodes $n(a)$ to the type of objects allocated at site a. When the analysis unifies two nodes, it takes the least upper bound (LUB) of the two types in the type hierarchy.

5.1.4 *Virtual Method Calls*

The presence of subtyping and virtual method calls raises two problems. From the point of view of the caller, the invoked method is not statically known. From the point of view of the callee, the number of region parameters being passed at the call site is not known, because the virtual methods it overrides may have different numbers of region parameters.

To solve the first problem, the analysis computes a points-to graph for each method m that incorporates the points-to information from all of other methods that override m. It does so by adding a dummy call from each method to the methods that immediately override it in the class hierarchy. During the analysis, the points-to information will transitively propagate from the methods at the bottom to the overridden methods at the top of the hierarchy. At a call site, the algorithm analyzes one single method, since the points-to information for that method incorporates the effects of all the methods that it may dispatch to.

For the second problem, that of mismatch between actual and formal region parameters for a method, the analysis creates an additional global points-to graph for each family of virtual methods (families are the equivalence classes induced by the overriding relation). That global graph merges together the information from all of methods in the family, providing a unique region space for formal parameters. The number of parameter regions for all calls to methods in the family is the same and is determined by this global graph. For each method, the analysis identifies a mapping from its formal parameters in the local space (given by the points-to graph of the method) into the formal parameters in the global space (given by the global points-to graph). This mapping allows virtual methods to select the regions they need from those that have been passed to them.

5.1.5 *Tracking Allocations and Accesses*

Tracking allocation and access effects during points-to analysis is fairly straightforward: we extend the abstraction to a tuple $G = (N, E, A, T)$, where $A \subseteq N$ is the set of allocation nodes and $T \subseteq N$ is the set of accessed nodes. The sets A and T represent all of the regions that the current method, including the methods it invokes, accesses or allocated objects into.

The analysis keeps track of the sets A and T as follows. The intra-procedural analysis marks each node $n(a)$ corresponding to an allocation site a as an allocation node. For each field access $x.f$ or method call $x.m(..)$, it marks the node $n(x)$ as an accessed node; and it marks each allocation node also as an accessed node. The unification of two nodes yields an allocation (or accessed) node if one of the two original nodes was an allocation (or accessed) node. At the inter-procedural level, for each allocation node n in the callee, the mapped node $m = \alpha(n)$ is an allocation node in the caller (and similarly for accesses).

5.2 Phase 2: Region Liveness Analysis

The next phase of the algorithm performs a flow-sensitive analysis to extract information about the live regions at each program point. We say that a region is live at point in the program if:

1. the region is reachable from at least one live variable at that program point; and
2. the execution of the program may access (i.e., read, write, or allocate objects into) the region in the future.

This section presents an analysis which computes liveness information for local regions only; Section 6.1 presents an extension that computes liveness information for parameter regions as well.

We formulate the algorithm as a backward dataflow analysis. The dataflow information that the algorithm computes is a pair of live regions and live variables. Let N_L be the set of local regions in the current method, defined as those regions that are not reachable from the parameter nodes in the points-to graph. Then, the dataflow information is: $(\mathit{Live}_r, \mathit{Live}_v) \in D = \mathcal{P}(N_L) \times \mathcal{P}(V)$. This forms a lattice domain whose meet operator is the component-wise set union, and whose top element is the pair of empty sets of live regions and variables: $\top = (\varnothing, \varnothing)$.

We define the transfer functions for statements as follows. Given the pair $(\mathit{Live}_r, \mathit{Live}_v)$ denoting the live regions and variables after a statement s, and the points-to graph $G = (N, E, A, T)$ of the current method, the liveness information $(\mathit{Live}'_r, \mathit{Live}'_v)$ before the statement is:

$$\mathit{Live}'_v = (\mathit{Live}_v - \mathit{def}_v) \cup \mathit{use}_v \quad (1)$$

$$\mathit{Live}'_r = (\mathit{Live}_r \cup \mathit{use}_r) \cap \mathit{Reach}_r \cap N_L \quad (2)$$

where the set Reach_r contains those regions that are reachable from live variables:

$$\mathit{Reach}_r = \{m \mid \exists y.\ y \in \mathit{Live}'_v \wedge \langle n(y), m\rangle \in E^*\} \quad (3)$$

and the sets def_v, use_v, and use_r are defined as:

Statement s	$\mathit{def}_v(s)$	$\mathit{use}_v(s)$	$\mathit{use}_r(s)$
$x = y$	$\{x\}$	$\{y\}$	$\varnothing$
$x = y.f$	$\{x\}$	$\{y\}$	$\{n(y)\}$
$x.f = y$	$\varnothing$	$\{x, y\}$	$\{n(x)\}$
$x = \texttt{new}\ C : a$	$\{x\}$	$\varnothing$	$\{n(a)\}$
$x = y_1.m(y_2, .., y_n)$	$\{x\}$	$\{y_1, .., y_n\}$	$\{n(y_1)\} \cup T^\alpha_m$
$\texttt{return}\ x$	$\varnothing$	$\{x\}$	$\varnothing$

The last column shows the regions that the program accesses. For the new statement, we use an explicit label $a \in S$ to describe the allocation site that the statement corresponds to. A statement accesses a region if it reads or writes a field of an object in the region ($x = y.f$ or $x.f = y$), if it creates a new object in the region ($\texttt{new}\ C$), or if it invokes the method of an object in the region ($y_1.m(y_2, \ldots, y_n)$). When invoking a method m we add those regions accessed by the callee: $T^\alpha_m = \{\alpha(n) \mid n \in T_m\}$, where T_m is the set of regions accessed by m and α is the mapping at this call site (as defined in Section 5.1.2).

It can be shown that the above transfer functions are monotonic. Finally, the dataflow information at the end of each method is the the set of parameter regions $N - N_L$. This completes the definition of a full dataflow framework for region liveness.

$$\frac{s \equiv (x = \texttt{new}\ C\ \texttt{in}\ r) \quad r \notin \mathit{Live}_r(\bullet s)}{\bullet s \implies \texttt{create}\ r}
\qquad
\frac{s \equiv (x = y_1.m\texttt{<}r_1,..,r_k\texttt{>}(...)) \quad r_i \notin \mathit{Live}_r(\bullet s)}{\bullet s \implies \texttt{create}\ r_i}$$

$$\frac{n \in \mathit{Live}_r(\bullet s) - \mathit{Live}_r(s\bullet)}{s\bullet \implies \texttt{remove}\ r_n}
\qquad
\frac{(p_1,p_2) \in \mathit{FlowEdges} \quad n \in \mathit{Live}_r(p_1) - \mathit{Live}_r(p_2)}{p_2 \implies \texttt{remove}\ r_n}$$

Figure 7: Rules for placement of region statements.

5.3 Phase 3: Region Translation

The final phase of the algorithm is the region transformation. For each method, the compiler assigns a region name r_n to each allocation node n in the points-to graph of that method. Then, it inspects each construct in the program and generates a corresponding region-annotated construct in the output program, as follows:

- *Object allocations.* It translates each statement "`new` C", whose allocation site is a, into: "`new` C `in` $r_{n(a)}$".
- *Method declarations.* Given a method m, the compiler inspects the graph of method m and identifies all allocation nodes that are reachable from formal parameters. Let these nodes be $n_1, \ldots, n_k$. The compiler then adds the regions $\texttt{<}r_{n_1}, \ldots, r_{n_k}\texttt{>}$ as formal region parameters of the method. Note that accessed nodes with no allocations, don't need to be passed as parameters.
- *Method calls.* At each call site, the compiler derives a mapping α between the formal parameters of the invoked method and the actual region parameters at the call site, using the algorithm from Section 5.1.2 (but the only rule from Figure 6 that applies at this point is rule (R7)). If $n_1, \ldots, n_k$ are the callee nodes corresponding to the formal parameters, the actual parameters at the call are: $r_{\alpha(n_1)}, \ldots, r_{\alpha(n_k)}$.

Finally, the compiler uses the region liveness information to place region creation and removal statements. We describe this process using rewriting rules of the form $p \Rightarrow c/r$, to denote that the compiler inserts a create or remove command c/r at program point p. For each statement s, we write $\bullet s$ for the program point before s, and $s\bullet$ for the program point after s.

Figure 7 shows the rewriting rules that describe the insertion of region creation and removal statements. The *FlowEdges* relation in the last rule models control-flow edges as pairs of program points (p_1, p_2), where program point p_2. The second premise of the rule can only be satisfied at control flow split points, because region liveness is a backward, "may" analysis.

Essentially, the compiler places region creation statements at the points where local regions become live, and removes them as soon as they become dead. A region becomes live when it is explicitly used at an allocation site or a call site, but is not live at those points. A region ceases to be live when it is live before a statement, but not after; or at control flow split points, when the region is live on one, but not all of the outgoing branches.

6. EXTENSIONS

This section presents several extensions to the algorithm from the previous section.

6.1 Inter-procedural Region Removal

The analysis presented so far uses a simple model to reason about regions in the program: it computes liveness information only for local regions in N_L. Regions passed in as parameters are assumed to be live in the callees. As a result, each region is always created and removed in the method where it is local. We present an extension which relaxes this constraint by computing liveness information for parameter regions as well. This extension enables the analysis to identify regions that can be removed early, in the invoked methods; hence, it identifies regions whose lifetimes span over multiple procedures.

The extended liveness algorithm works as follows. For each method m, it keeps track of a set D^m of region parameters that are dead at the end of method m and unaliased with any other region parameter. Initially, all of these sets optimistically contain all the parameter regions: $D^m = N^m - N^m_L$, where N^m, N^m_L denote the sets N, N_L for method m.

First, the algorithm performs an intra-procedural region liveness analysis for each method m, as presented in Section 5.2, but computes liveness information for all regions in N^m, not only for local regions in N^m_L. For this, we change the filtering in Equation (2) to:

$$\mathit{Live}'_r \;=\; (\mathit{Live}_r \,\cup\, \mathit{use}_r) \,\cap\, \mathit{Reach}_r \,\cap\, (N_L \cup D^m)$$

Then, the algorithm uses a fixed point approach to analyze call sites and determine if the optimistic assumption is violated. If so, it removes the violating regions from the sets D^m. More precisely, the algorithm inspects, for each method m, all of the calls to m, in particular, each formal parameter in D^m and the corresponding actual region in a caller method m'. If the actual region is live or is passed as another formal parameter in the same call, then the analysis removes that parameter from D^m and re-analyzes all of the methods that m invokes. Formally:

$$\frac{s \equiv (x = y_1.m\texttt{<}r_1,..,r_i,..,r_n\texttt{>}(...)) \quad p_i \in D^m \qquad m\ \texttt{<}p_1,..,p_i,..,p_n\texttt{>}(...)\{...\} \qquad r_i \in \mathit{Live}^{m'}_{ip}(s\bullet) \ \vee\ \exists j \neq i\,.\,r_i = r_j}{D^m \ \leftarrow\ (D^m - \{p_i\})}$$

where the set $\mathit{Live}^{m'}_{ip}(s\bullet)$ is the set of live regions derived using current intra-procedural liveness results $\mathit{Live}^{m'}_r$ and the current values in the set $D^{m'}$:

$$\mathit{Live}^{m'}_{ip}(s\bullet) \;=\; \mathit{Live}^{m'}_r(s\bullet) \,\cup\, (\ N - (N_L \cup D^{m'})\ )$$

The condition $\exists j \neq i\,.\,r_i = r_j$ in the premise of this rule disallows the analysis to remove a parameter region that is aliased with another region; otherwise, the region could be removed multiple times. Once a region is aliased at a call site (i.e., passed as parameter twice), it is marked as live, so neither the callee, nor the methods that the callee invokes can remove the region.

This above algorithm can be implemented efficiently by traversing the strongly connected components of the call graph in (direct) topological order and analyzing each component once.

6.2 Stack Allocation

The algorithm can be easily extended to enable the stack allocation of objects. The analysis must identify regions which hold a statically bounded number of objects. If the following conditions hold for a local region, then the compiler can allocate it on stack:

- *No escaped allocations.* Whenever the region is passed as parameter to another method, the callee doesn't allocate new objects in the region. We allow regions to escape the current method, as long as the invoked methods do not allocate new objects in them.
- *Bounded allocations.* Between the creation and removal of the region in the current method, there must be a bounded number of objects allocations in the region. We use a simple dataflow analysis that computes a set of live allocations $a \in S$ at each statement. If an allocation a is never live at the point where it is executed, then there is a bounded number of objects that this allocation site places into the region.
- *Array allocations.* Arrays allocated in the region must have a statically known size.

6.3 Multithreading

Because our points-to analysis is flow-insensitive, it conservatively characterizes multithreaded executions. The region liveness analysis, however, is flow-sensitive and is not sound in the multithreaded setting: a region local to a method may escape to a child thread and outlive that method. The single-threaded analysis will incorrectly assume that the method outlives all of its local regions, and that these regions can be removed when the method returns.

To ensure a correct program execution, we use an approach that requires a cooperation between the analysis and the run-time region system. More precisely, the analysis of each thread works as presented so far, with the assumption that local regions do not outlive their methods. When regions escape from a parent thread to a child thread, the analysis marks them as dead at the end of the child thread. We use a run-time solution based on reference counts (similar to the approaches in [16, 6]) where each region keeps a counter that indicates how many threads concurrently access it. When a thread is forked, the program increments the counter of all the regions passed to the child thread. When the parent or the child attempts to remove the region via a `remove` statement, it decrements the counter. The region gets actually removed only if the counter becomes zero. Thus, the last thread that accesses the region is the one that removes it.

Multithreading also requires a number of changes in the analysis. First, a parent thread must pass to the child thread all of the regions it accesses, not just the allocation regions; otherwise, regions may be unsafely deallocated by the parent thread while the child still accesses them. Therefore, the program must pass these regions as parameters from their creation points to the point where the child thread is being forked. For this, the analysis marks all of the regions accessed by child threads as *shared*, and tracks the shared flags during the interprocedural analysis (similar to the way it tracks access and allocation flags). In the translation phase, the analysis passes as region parameters both the allocation regions and the shared regions. Next, the analysis must account for synchronization operations: whenever the program acquires or releases a lock on object x, it marks node $n(x)$ as accessed; similarly, when the program synchronizes on x, the liveness analysis adds $n(x)$ to the set of live regions. Finally, to avoid placing shared objects on the stack of one particular thread, the compiler must require an additional stack allocation condition in the previous section: that stack allocated regions must not be shared.

6.4 Exceptions

Exceptions raise two issues in the context of region analysis. First, when an exception occurs, the exception object being thrown escapes the procedure and the analysis cannot compute its lifetime. Therefore, the analysis marks thrown objects as immortal and does not allocate them in regions.

Second, if a method terminates abruptly with an exception, the program must reclaim all of the local regions that are still live. Our solution relies on run-time support. When an exception occurs, the exception run-time system that walks up the stack deallocates all of the local regions for each method whose activation frame is traversed. One complication is that the run-time system must determine the subset of local regions that are live, and just remove those. A simpler solution is to use an additional level of indirection for regions such that, if the program attempts to remove a region multiple times, it only removes the region the first time and has no effect afterward. This enables the run-time system to remove all of the local regions of methods when it walks up the stack, regardless of how many of them have already been removed.

7. EXPERIMENTAL RESULTS

We have implemented the algorithm and all of the extensions presented in this paper as a bytecode-level transformation in the Soot infrastructure [24]. Our compiler takes an input Java program and generates an equivalent region-annotated program. We have extended the standard Java bytecodes with additional opcodes to support the region operations. We have also extended the interpreter of the KaffeVM [26] to support the new bytecodes and provide the run-time mechanisms for manipulating regions, including the extensions discussed in Section 6. We added instrumentation code in the extended virtual machine to collect region statistics. We ran the experiments on a 2 GHz Pentium machine running KaffeVM 1.07 on Linux.

7.1 Analysis Measurements

Table 1 presents our list of benchmark programs, which consists of the Java Olden benchmarks [7]. The first two columns in the table show the application size before and after the transformation, and indicate that the region annotations introduce little space overhead. Next, the table shows the number of methods that our system analyzes. These numbers include all of the library methods that applications invoke. Finally, the last two columns in the table show the running times of our analysis, which includes points-to analysis and region liveness analysis; and the total compilation time, which includes loading the input files, building the Soot representation, and writing the output files. The running times indicate that the analysis is tractable in practice and represents only a fraction (16% on average) of the total compilation time.

Program	Size(Kb) Original vs.	Translated	Methods analyzed	Time (sec) Total vs.	Analysis
bh	31.1	32.8	916	23.7	3.8
bisort	8.5	9.1	825	21.9	3.3
em3d	12.6	13.6	871	22.8	3.7
health	16.8	17.7	844	22.3	3.4
mst	12.0	12.6	841	22.0	3.3
perimeter	15.6	16.2	858	22.3	3.3
power	26.4	27.2	879	23.0	3.5
treeadd	5.4	5.8	816	21.7	3.2
tsp	11.3	12.0	828	22.0	3.3
voronoi	27.1	29.1	928	23.9	4.5

Table 1: Benchmarks and analysis times

Program	Maximum size (Kb) None	GC	Region	Savings w.r.t. None	GC
bh	25,852	2,682	3,076	88%	-15%
bisort	413	412	412	0%	0%
em3d	3,446	3,412	3,445	0%	-1%
health	22,206	3,714	6,450	71%	-74%
mst	33,122	33,114	33,101	0%	0%
perimeter	14,580	14,572	14,563	0%	0%
power	22,030	2,923	749	97%	74%
treeadd	21,057	21,049	21,040	0%	0%
tsp	13,193	6,481	5,851	56%	10%
voronoi	16,764	8,152	16,369	2%	-101%

Table 2: Maximum memory usage

7.2 Memory Measurements

Table 2 presents the maximum memory utilization during the execution of these benchmarks. We compare these numbers in three settings: running the applications with no memory management (first column), running them with automatic garbage collection (second column), and running them with region-based support (third column). The values given for the garbage-collected run were measured using the default KaffeVM incremental garbage collector. Since the GC system reclaims memory dynamically and the region-based system statically decides when to reclaim it, the comparison to the GC system is meant to provide an upper bound for the possible savings, rather than a competition for the region-based system.

The numbers collected only include the application memory size, not the memory used by the virtual machine itself. The first three columns in Table 2 show the maximum amount of memory required during the execution of the program (i.e., the memory watermark) in each of the three settings. The following two columns project these numbers into percentages: they show the relative memory savings of the region-based system with respect to the "None" setting (column four) and to the GC setting (column five).

These numbers show that the region-based system can yield significant absolute memory savings for several benchmarks; and that for most of the benchmarks with no absolute savings, the GC system can't save memory either.

Surprisingly, for two of the benchmarks, *power* and *tsp*, the region-based system outperforms the GC system. In particular, for *power*, the region-based system uses 4 times fewer memory than the GC system. A closer inspection of this situation revealed that the garbage collector has not been triggered often enough by Kaffe, since this application uses very little memory. Once we forced collections to be performed frequently enough, the GC system yielded roughly the same numbers as the region-based system.

For a number of other benchmarks, all three systems use roughly the same amount of memory, more precisely for *bisort*, *em3d*, *mst*, *perimeter* and *treeadd*, showing that neither static, nor dynamic techniques can reclaim memory and most data objects are long-lived. For *bh*, *health*, and *voronoi*, the GC system performs better, showing the limitations of static analysis. Even in such cases, the region-based system may bring significant benefits over a system with no memory management, as in the case of *bh* and *health*, where our transformation saves respectively 88% and 71% of the memory compared to "None".

7.3 Region Statistics

Table 3 shows several region statistics. The first two columns represent the percentage of data allocated on stack and in regions throughout the execution of the program. The remaining percentage is the immortal data. The third column shows the average lifetime of all regions and stack-allocated data, weighted by their sizes. These numbers indicate that the analysis is usually able to allocate a large fraction of the data on stack or in regions, but the average lifetime varies with the application. The lifetimes range from small values such as 1% for *bh* to 100% for *perimeter*.

As expected, there is a correlation between the numbers in this table and the memory savings from Table 2: our system yields absolute memory savings whenever it places most of the data on stack or in short-lived regions. For instance, for *mst* and *em3d* the analysis places almost no data in regions or stack, so there are no savings; and for *bh* or *power*, it places almost all of the data on stack or in short lived regions, and memory savings are substantial.

The last two columns in Table 3 show how many regions are being manipulated during program execution. Column four shows how many regions have been created throughout the entire execution, and column five shows how many regions are live at any given time. These numbers indicate that there is a lot of fluctuation in the total number of regions, ranging from less than hundred regions to more than a million regions. The maximum number of regions at each moment is usually low, with one exception, *health*; even then, the number is only a fraction of the total number of regions. In general, this table suggests that, for programs where most of the data gets placed in regions, larger total numbers of regions are correlated with lower region lifetimes and with larger memory savings (e.g. for *bh*, *health*, or *tsp*).

7.4 Discussion

As expected, our results indicate that the ability of the region-based system to save memory depends on memory characteristics of each particular application. The memory watermark and savings, the ability to allocate data on stack, the average region lifetimes, and total or maximum numbers of regions all vary with the application and fluctuate across our set of benchmarks, ranging from very small to very large values.

However, the overall results for our set of benchmarks are encouraging: the region-based system was able to save significant amounts of memory for most of the cases where data was not live throughout the program. In fact, although

Program	Mem. Distrib. Stack	Regions	Average Lifetime	Dyn. Regions Total	Max.
bh	22%	66%	1%	478057	2446
bisort	0%	80%	99%	81	29
em3d	0%	0%	23%	88	28
health	22%	78%	30%	1023269	170035
mst	0%	0%	21%	1080	30
perimeter	0%	99%	100%	81	28
power	97%	1%	1%	80	29
treeadd	0%	99%	99%	81	28
tsp	0%	99%	44%	262223	29
voronoi	0%	96%	97%	24657	54

Table 3: Region Statistics

it not intended to compete with the run-time GC system, our statically-enabled region-based system performed about as good as, or even better than the GC system for most of the benchmarks (8 out of 10 benchmarks).

Finally, our experiments suggest that our region analysis and transformation is a good match for real-time systems. First, our results show that the transformation can be successful at saving memory, while avoiding pauses for unbounded amounts of time during program execution. Second, because our analysis techniques are sound, programs can perform region manipulations safely and efficiently, without the run-time overhead that would be otherwise required to ensure the absence of accesses to deallocated data. And third, since our system produces all of the region annotations automatically, our system frees programmers of the burden of writing such annotations: users can write real-time applications without worrying about memory reclamation, and have our system generate the appropriate region annotations that would make such programs suitable to a real-time environment.

8. RELATED WORK

Tofte and Talpin [23, 2] propose a region inference algorithm for a simply typed lambda calculus. Regions are lexically scoped in their calculus, which imposes a stack discipline on the region lifetimes. In [2] they introduce a region resetting operation to alleviate the limitations of the original system. Our system can automatically model region resetting using the region creation and removal primitives, as shown in the example from Section 3.

Other approaches build on the Tofte/Talpin algorithm, but lexically decouple region allocation from region deallocation [1, 16]. In particular, Aiken, Fahndrich, and Levien (AFL) [1] propose an inference system with the same goals as ours: compute region liveness information to automatically place region creation and removal constructs at different points in the program. They formulate the analysis using a system of custom constraints specifically designed for identifying region lifetimes in a functional setting, and focus on features such as polymorphism and higher-order functions. In contrast, our analysis is designed to identify region liveness in an imperative setting, using techniques closer to standard live variable analysis. Our analysis focuses on imperative control-flow, pointers, destructive updates, object-oriented, and multithreaded language constructs, none of which are addressed by [1]. Furthermore, the AFL system is less general: it cannot identify loop-carried region lifetimes, since regions cannot be re-allocated once they have been deallocated.

Lattner and Adve [17] propose pool allocation for C programs. They also use a pointer analysis similar to [18], and present a transformation which allocates/deallocates regions at procedure entry/exit points. However, their system cannot identify arbitrary (e.g., loop-carried or inter-procedural) region lifetimes, doesn't handle object-orientation, and has not been applied to the memory management problem.

In work performed concurrently with ours, Chin et.al. [8] propose a region inference system for an object-oriented core calculus. Their system translates programs into an output calculus with a provably sound region type system. They use lexically scoped regions and forbid dangling pointers; hence, their approach is less precise and flexible than ours. Furthermore, they do not handle multithreading or exceptions, and do not provide experimental data regarding the efficiency of the generated programs.

Other researchers have explored the dynamic detection of regions in Java programs [19, 11], or have proposed language support for regions in C [12, 13, 15] or Java [10, 6]. These techniques are orthogonal to the static detection of regions presented in this paper. Possible directions of research include developing hybrid analyses that combine static and dynamic techniques, or using our analyses to provide default region annotations in languages with region support.

In the area of pointer analysis, the relevant work is that based on unification constraints, originally proposed by Steensgaard [22]. Subsequent work has extended this algorithm with context-sensitivity for C programs [18, 17]. These analyses are similar to the one presented in this paper, but describe the call site analysis informally or refer to it as "inlining graphs". Our work provides a clear, formal specification of the analysis at call sites and shows how to apply the analysis to the region inference and memory management problems.

Finally, related research has explored combining pointer and escape analysis for Java programs [3, 9, 25, 4, 14, 21] to identify when objects escape from the methods or threads that create them. Such analyses can enable the allocation of objects on stack (when objects don't escape their enclosing method) or the removal of synchronization operations (when object don't escape their current thread). Our system gives a solution to a more general problem: instead of identifying whether or not object lifetimes match method lifetimes, we compute unrestricted object lifetimes and use a region-based system to reclaim memory according to the computed lifetimes.

9. CONCLUSIONS

We have presented a region analysis and transformation framework for Java programs. Given an input program, the algorithm generates an equivalent output program with explicit region-based memory management support. This transformation uses points-to analysis to capture the aliasing information in the program and uses novel analysis algorithms to compute region liveness information. Our experimental results indicate that this approach is able to place a large fraction of the objects in regions or on stack, that many regions are short-lived compared to that of the whole program, and that the region-based system can yield significant memory savings for some of the applications.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions and comments.

10. REFERENCES

[1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[2] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

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

[4] 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.

[5] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, Mass., 2000. `http://www.rtj.org/`.

[6] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the SIGPLAN '03 Conference on Program Language Design and Implementation*, San Diego, CA, June 2003.

[7] Brendon Cahoon. The Java Olden benchmarks. `www-ali.cs.umass.edu/~cahoon/olden`.

[8] W. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *Proceedings of the SIGPLAN '04 Conference on Program Language Design and Implementation*, Washington, DC, June 2004.

[9] 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.

[10] M. Christiansen and P. Velschow. Region-based memory management in Java. Master's thesis, DIKU, University of Copenhagen, May 1998.

[11] M. Deters and R. Cytron. Automated discovery of scoped memory regions for Real-Time Java. In *Proceedings of the 2004 International Symposium on Memory Management*, Berlin, Germany, June 2002.

[12] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[13] 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.

[14] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the 2000 International Conference on Compiler Construction*, Berlin, Germany, April 2000.

[15] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[16] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, Florence, Italy, September 2001.

[17] C. Lattner and V. Adve. Automatic pool allocation for disjoint data structures. In *Proceedings of The Workshop on Memory Systems Performance*, Berlin, Germany, June 2002.

[18] D. Liang and M.J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the ACM SIGSOFT '99 Symposium on the Foundations of Software Engineering*, Toulouse,France, September 1999.

[19] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. In *Proceedings of the 2004 International Symposium on Memory Management*, Berlin, Germany, June 2002.

[20] R. Rugina and S. Cherem. Region analysis for imperative programs. Technical Report CS TR2003-1914, Cornell University, October 2003.

[21] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, Utah, June 2001.

[22] 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.

[23] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 1994.

[24] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON '99*, Toronto, Canada, November 1999.

[25] 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.

[26] Tim Wilkinson. Kaffe – a free Java virtual machine. `http://www.kaffe.org`.