Identification of Logically Related Heap Regions

Mark Marron[1] Deepak Kapur[2] Manuel Hermenegildo[1]

[1]IMDEA-Software (Madrid, Spain)
[2]University of New Mexico (New Mexico, USA)
{mark.marron, manuel.hermenegildo}@imdea.org, kapur@cs.unm.edu

Abstract

This paper introduces a novel set of heuristics for identifying logically related sections of the heap such as recursive data structures, objects that are part of the same multi-component structure, and related groups of objects stored in the same collection/array. When combined with lifetime properties of these structures, this information can be used to drive a range of program optimizations including pool allocation, object co-location, static deallocation, and region-based garbage collection. The technique outlined in this paper also improves the efficiency of the static analysis by providing a compact normal form for the abstract models (speeding the convergence of the static analysis).

We focus on two techniques for grouping parts of the heap. The first is a technique for identifying recursive data structures in object-oriented programs based on connectivity and type information. The second technique is a method for grouping objects that make up the same composite structure and that allows us to partition the objects stored in a collection/array into groups based on a *similarity* relation. We provide a parametric component in the *similarity* relation to support specialized analysis applications (e.g. numeric analysis of object fields). Using the Em3d and Barnes-Hut benchmarks from the JOlden suite we show how these grouping methods can be used to identify various types of logical structures and enable the application of many region-based optimizations.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages (program analysis)

General Terms Languages, Performance, Verification

1. Introduction

Identifying and grouping logically related parts of the program heap in an abstract program model is useful both to client optimization applications (which can use the information to perform pool allocation, object co-location, static deallocation, etc.) and in improving the performance of the static data flow analysis (providing a normal form which speeds the convergence of the analysis). This paper presents a novel set of grouping heuristics for identifying and grouping these regions in a manner that supports a wide range of client applications and that can be used in practice to produce an efficient static analysis.

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

ISMM'09 June 19–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-347-1/09/06. . . $5.00

Research on object allocation and memory layout has used the notions of logically related structures to improve the spatial locality of objects with similar temporal accesses via techniques such as pool allocation [16, 4] and object co-location [12, 8]. Other applications which use logically related sections of the heap have focused on improving the efficiency of garbage collection. The most direct application is static deallocation of regions or data structures [16, 5, 13]. There has also been work [15] on using region information to reduce the pause times of garbage collection by only performing the collection on portions of heap that are likely to contain many dead objects. Similar approaches (when combined with heap based read/write information) can also be used to support parallel garbage collection by statically identifying which parts of the heap can be safely collected without concern for the mutator.

The techniques listed above use a variety of approaches for identifying region information that is later used in the optimization phase. The techniques range from simple grouping via the points-to partitions computed using a Steensgaard style analysis [26, 14] to more sophisticated approaches as done in [16, 17, 13]. However, as these techniques are based on points-to style analyses or use limited amounts of context/flow sensitivity they cannot precisely model many properties (sharing and shape) of data structures that are used extensively in object oriented programs. The technique in this paper offers a significantly higher degree of precision for identifying regions than these approaches and can be used directly to improve the effectiveness of many region based memory optimizations.

In addition to being useful for a range of optimization techniques the region identification technique we present in this paper can be used to improve the performance of various static analysis techniques. This is achieved by using the region identification to define a normal form for the abstract models, reducing the height of the abstract lattice. This use of a normal form can be seen as a pseudo-widening operation used to transform a domain of infinite height (e.g, linked lists of size $0, 1 \ldots \infty$) into a finite height lattice (e.g, linked lists that are of size $0, 1, 2,$ or some unknown length ω). There are two parts to this normalization that we address in this paper. The first is the compression of recursive structures of potentially unbounded size, such as lists or trees, into finite representations. The second is the grouping of objects that make up composite data structures or partitioning objects stored in a collection/array based on a *similarity* relation.

While the concept of computing normal forms for heap representations is not novel to this paper —symbolic access paths in [7], normalization/merge in [19, 18, 3, 6, 21], and the append left/right rules in [1, 28] or similar rules for inductive synthesis in [11]— the heuristics we use to accomplish this are significantly more general than what has been used in previous work. In particular the formalization applies to any type of recursive data structure (as opposed to just lists or trees [1, 28, 19, 11]) and recursive data structures that are part of larger composite structures (such as in [1, 21]). The heuristics in this work can precisely model multi-

component structures with shared components which cannot be handled by [1, 28, 19]). Finally these heuristics support more effective grouping of the contents of collections (arrays or collections from `java.util`) than is possible with the methods described in [3, 23, 10] (and which are left out in most other approaches).

We begin with a brief introduction of the parametric labeled storage shape graph (*lssg*) model, Section 2, that we use to illustrate the main contributions of this paper. These contributions as described in Sections 3 and 4 are:

- A method for identifying and grouping recursive data structures.
- A method for grouping objects that form multi-object composite structures.
- A parametric approach to grouping the contents of arrays/collections.

Finally, in Section 6 we use the Em3d and Barnes-Hut benchmarks from the well known JOlden suite to illustrate the results of the region analysis and how this information can be used to support some of the optimizations mentioned above.

2. Concrete Heap and Labeled Shape Graph

We begin by reviewing the abstract graph model that we build on in this work (although the concepts presented in this paper can be applied in other approaches such as those that rely on separation logic [1, 11, 28]). In previous work [21, 22, 23] this model is used to precisely perform shape and sharing analysis on a range of Java programs. While the properties discussed therein are critical to precisely analyzing these programs (and similarly the region identification method presented in this paper is critical to the results in these papers), we do not need all of this information in order to perform region identification and grouping. Thus, to simplify the discussion and to focus on the novel concepts in this paper we present a simplified version of the model.

2.1 Concrete Semantics

To analyze a program we first transform (via a modified compiler frontend) the Java 1.4 source into a semantically equivalent program in a simpler to analyze core language. This intermediate language is statically typed, has method invocations, conditional constructs, exception handling and the standard looping statements. The state modification and expressions cover the standard range of program operations: load, store, and assignment along with logical, arithmetic, and comparison operators. During this transformation step we also load in our specialized standard library implementations, so we can analyze programs that use classes from `java.util`, `java.lang`, and `java.io`.

The semantics of memory are defined in the usual way, using an environment, mapping variables into values, and a store, mapping addresses into values. We refer to the environment and the store together as the concrete heap, which is treated as a labeled, directed multi-graph (V,O,R) where each $v \in V$ is a variable, each $o \in O$ is an object on the heap and each $r \in R$ is a reference (either a variable reference or a pointer between objects). The set of references $R \subseteq (V \cup O) \times O \times L$ where L is the set of storage location identifiers (a variable name in the environment, a field identifier for references stored in objects, or an integer offset for references stored in arrays/collections).

A *region* of memory $\Re = (C,P,R_{in},R_{out})$ consists of a subset $C \subseteq O$ of the objects in the heap, all the pointers $P = \{(a,b,p) \in R \mid a,b \in C \wedge p \in L\}$ that connect these objects, the references that enter the region $R_{in} = \{(a,b,r) \in R \mid a \in (V \cup O) \setminus C \wedge b \in C \wedge r \in L\}$ and references exiting the region $R_{out} = \{(a,b,r) \in R \mid a \in C \wedge b \in O \setminus C \wedge r \in L\}$.

2.2 Storage Shape Graph Abstraction

Our abstract heap domain is based on the *storage shape graph* [3] approach. An *abstract storage graph* is a tuple of the form $(\hat{V},\hat{N},\hat{E})$, where $\hat{V}$ is a set of abstract nodes representing the variables, $\hat{N}$ is a set of abstract nodes (each of which abstracts a region $\Re$ of the heap), and $\hat{E} \subseteq (\hat{V} \cup \hat{N}) \times \hat{N} \times \hat{L}$ are the graph edges, each of which abstracts a set of pointers, and $\hat{L}$ is a set of abstract storage *offsets* (variable names, field offsets or the special offset *?* for references stored in arrays/collections). We extend this definition with a set of additional relations $\hat{U}$ that further restrict the set of concrete heaps that each shape graph abstracts. The *labeled storage shape graphs (lssg)*, which we refer to simply as *abstract graphs*, are tuples of the form $(\hat{V},\hat{N},\hat{E},\hat{U})$.

DEFINITION 1 (Valid Concretization of a *lssg*). *A given concrete heap h is a* valid concretization *of a* labeled storage shape graph *g if there are functions Π_v, Π_o, Π_r such that the following hold:*

- $\Pi_v : V \mapsto \hat{V}$, $\Pi_o : O \mapsto \hat{N}$ *and* $\Pi_r : R \mapsto \hat{E}$ *are functions (and* Π_v *is 1–1).*
- $h, \Pi_v, \Pi_o,$ *and* Π_r *satisfy all the relations in* $\hat{U}$.
- $h, \Pi_v, \Pi_o,$ *and* Π_r *are* connectively consistent *with g.*

Where h, Π_v, Π_o, Π_r *are* connectively consistent *with g if:*

- $\forall\, o_1, o_2 \in O$ *s.t.* $(o_1,o_2,p) \in R$, $\exists\, e \in \hat{E}$ *s.t.* $e = \Pi_r((o_1,o_2,p))$, *e starts at* $\Pi_o(o_1)$*, ends at* $\Pi_o(o_2)$*, and* e.offset $= p$.
- $\forall\, v \in V$, $o \in O$ *s.t.* $(v,o,v) \in R$, $\exists\, e \in \hat{E}$ *s.t.* $e = \Pi_r((v,o,v))$, *e starts at* $\Pi_v(v)$*, ends at* $\Pi_o(o)$*, and* e.offset $= v$.

To check if a given concrete heap h and maps Π_v, Π_o, Π_r satisfy a given relation in $\hat{U}$ we need to look at the pre-images of the nodes and edges in the abstract graph g under the maps Π_v, Π_o, Π_r. We use the notation $h \downarrow_g e$ to indicate the set of references in the concrete heap h that are in the pre-image of e under the maps. Similarly, we use $h \downarrow_g n$, to indicate the region of the heap that is the pre-image of n under the maps.

2.3 Label Relations (in $\hat{U}$)

Type. For the *type* relation, we add a relation $(n, \{\tau_1, \ldots, \tau_k\})$ (we use the shorthand $n.type = \{\tau_1, \ldots, \tau_k\}$) to $\hat{U}$ for each node in $\hat{N}$, where τ_j are types in the program and say: h, Π_v, Π_o, Π_r satisfies $(n, \{\tau_1, \ldots, \tau_k\})$ iff $\{\mathtt{typeof}(o) \mid \text{object } o \in h \downarrow_g n\} \subseteq \{\tau_1, \ldots, \tau_k\}$.

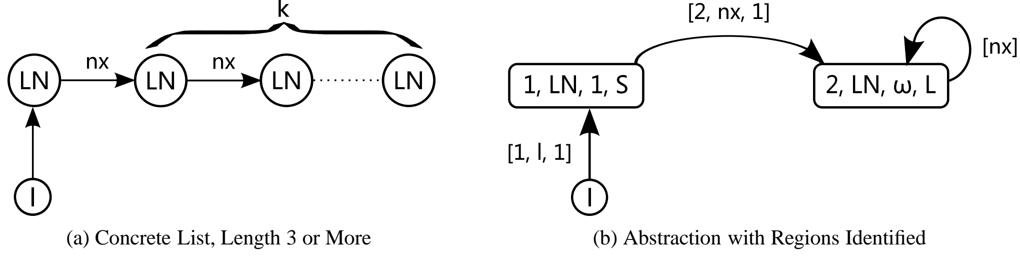
Linearity. The *linearity* relation is used to track the number of objects in the region abstracted by a given node or the number of references abstracted by a given edge. The *linearity* property has 2 values: 1 indicating a cardinality of $[0,1]$ or ω indicating a cardinality of $[0,\infty)$. Given a node n where $h \downarrow_g n = (C,P,R_{in},R_{out})$ then:

$$|C| \in \begin{cases} [0,1] & \text{if } n.linearity = 1 \\ [0,\infty) & \text{if } n.linearity = \omega \end{cases}$$

Similarly for an edge e where $h \downarrow_g e = \{r_1, \ldots, r_j\}$ then:

$$|\{r_1, \ldots, r_j\}| \in \begin{cases} [0,1] & \text{if } e.linearity = 1 \\ [0,\infty) & \text{if } e.linearity = \omega \end{cases}$$

Abstract Layout. To approximate the shape of the structures present in the region that a node abstracts, the analysis uses *abstract layout* properties $\{$*(S)ingleton*, *(L)ist*, *(T)ree*, *(D)ag*, and *(C)ycle*$\}$. The *(S)ingleton* property states that there are no pointers between any of the objects abstracted by the node, given a node n where $h \downarrow_g n = (C,P,R_{in},R_{out})$ then $P = \emptyset$. The other properties correspond to the standard definitions for list, tree, DAG, and cyclic structures in the literature [9, 21, 1].



(a) Concrete List, Length 3 or More

(b) Abstraction with Regions Identified

Figure 1: A Linked List and Desired Abstraction with Regions Identified

2.4 Sample Heap and Abstract Graph Model.

Figure 1 shows a linked list of length 3 or more (left) and the representation of this list in the abstract domain with the objects that represent it grouped into regions (right). In the abstract domain each edge is labeled with a unique identifier, an abstract storage *offset*, and a *linearity* label. The nodes are labeled with a unique identifier, a *type* label, a *linearity* label and a *layout* label.

In Figure 1b we see that the variable `l` refers to node 1 which represents a single (*linearity* is 1) ListNode (`LN`) object at the head of the linked list. There is a single edge (edge 2) out of the node representing the single (again *linearity* 1) `nx` (next) pointer, which ends at node 2. This node represents the tail of the list (the self `n` edge and *(L)ist layout*) which may contain many objects (*linearity* is ω).

Partitioning the list into these two nodes captures several important attributes. First we have kept the head of the list (which may be modified though the variable `l`) distinct, giving more opportunities to the analysis for precisely modeling the effects of later program statements. Next, the grouping has produced a compact representation for the list structure which has a substantial impact on the efficiency of the analysis. Finally, we have grouped all of the objects that make up the list into two nodes (the head and the tail, nodes 1 and 2) and as we will see later if there are other unrelated lists in the program (and the analysis can determine that they are unrelated) the abstraction will generate separate nodes for each of these lists. Thus, the information needed by the various optimization techniques we are interested in is preserved (objects in the same structures are grouped together while disjoint structures in the concrete heap are kept separate in the abstract model).

3. Recursive Components

The first contribution in this paper is a generalized method for identifying parts of the abstract heap graph that may represent a single recursive data structure and how these parts should be grouped together (e.g. using multiple nodes to represent the head and tail sections of the linked list). The basic approach of identifying potentially recursive structures is a straightforward examination of the type information and connectivity properties of the program based on recursive field paths [7, 21, 1, 19]. However, there are a number of subtle but important modifications that are needed to maintain the desired level of precision in the results when dealing with non-trivial object-oriented programs.

3.1 Statically Recursive Types

We can identify the types in a program that may be recursive by looking at the type graph for the program. This *static program type graph* has a node for each type that is declared and for each pair of types τ, τ' there is an edge from τ to τ' if τ has a field of type (or supertype) τ'. From this construction we can identify types that are recursive (based on the static type information) as follows:

DEFINITION 2 (Statically Recursive Types). *For a given program and types τ, τ':*

1. *τ, τ' are* statically recursive *iff in the* static program type graph *($\tau \neq \tau' \wedge \tau, \tau'$ are in the same strongly connected component) $\vee$ ($\tau = \tau'$ and there is a self edge).*
2. *τ is a* statically recursive *type iff $\exists \tau'$ s.t. τ, τ' are* statically recursive.

In much of the past work on region identification [21, 7, 19, 1, 11] this static type information has been used (in various ways) to determine if two objects are part of the same recursive data structure. However, this can result in overly approximate region identification in three important classes of heap structures. Below we describe these and how we can modify our concept of recursive structures to characterize them.

3.2 Safe Nodes

In order to accurately simulate the effects of various program statements it is critical to precisely model the targets of variable references. Consider removing an element from a linked list where we have multiple variables pointing into the same list structure. In order to preserve the listness property after the removal we must keep track of the relative positions of the variable references into the list structure and the effects of the assignment statements on the objects referred to by the variables. Thus, even though all of these objects make up the same recursive list structure, we want to use multiple nodes to represent it (one for each location in the list that is being modified in addition to nodes for the tail or other segments).

To identify these important objects which need to be modeled independently we introduce the notion of *safe nodes* (which is similar to the notion of *interrupting nodes* in [19]). We say a node is safe if it represents an interesting point in a recursive data structure (a point where the program is accessing a specific node in the data structure via a variable, as in the above example, or a non-recursive data structure pointing into specific locations in the recursive structure) and we keep these nodes distinct from any other recursive components.

If we have a recursive data structure and we store references to important points in it via another data structure we want to be able to maintain the relations between these specific points in a data structure. This is a generalization of maintaining the precise locations of variable references into a recursive data structure. This is important to analyzing situations of the form: a method returns a `Pair` object containing two references to `ListNode` objects and we want to remove all the elements in the list between the `first` and `second` entries of the `Pair`. If the analysis does not maintain the order relation between the targets of the `first` and `second` reference fields in the list structure we cannot accurately model the effects of the remove operation (e.g., we would conservatively assume that the target of the `second` field could come before the target of the `first` field in the list).

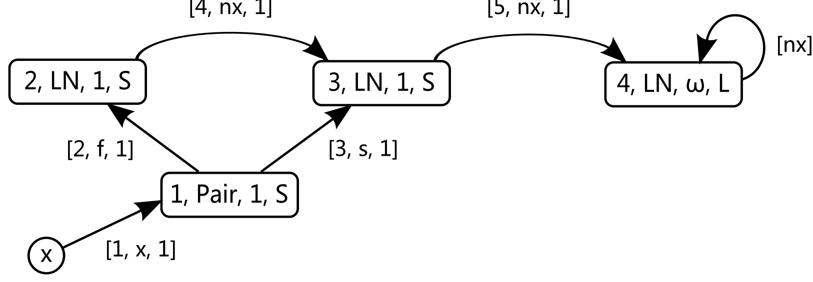


Figure 2: Safe Nodes Example

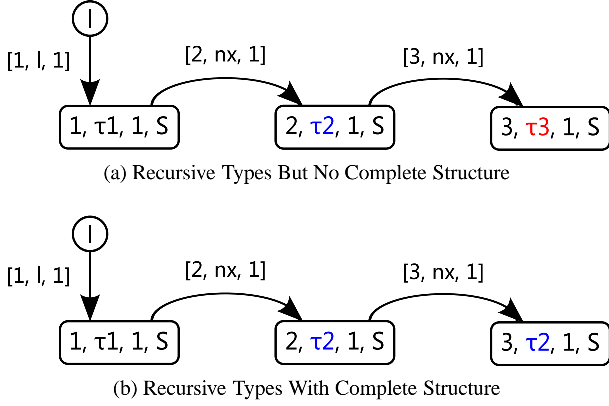


Figure 3: Recursive Types and Complete Structures

DEFINITION 3 (Safe Node). *A node n is* safe *if it is a node with the* (S)ingleton layout *and either of the following hold:*

1. *$\exists$ variable v that refers to n.*
2. *$\exists$ edge e s.t. e starts at a node n_s where $\forall \tau_s \in n_s.\text{type}, \tau \in n.\text{type}$, τ_s, τ are not* statically recursive.

Figure 2 shows a simple example of the two ways a node is considered safe (represents an interesting point in the heap). In this figure we have node 1 which is *safe* since it is referred to directly by a variable. More interestingly we have nodes 2, 3 which both represent `LN` objects and are *statically recursive* but are also pointed to by the `Pair` object which is *not statically recursive* with the `LN` type. Thus according to our definition of safe nodes, nodes 1, 2 are considered safe and will not be merged.

3.3 Connectivity Awareness

Consider a program with the object types $\tau 1$, $\tau 2$, $\tau 3$ which are mutually recursive on the `nx` field. If we have the abstract heap graph in Figure 3a we can see that the 2nd and 3rd nodes in the list are statically recursive according to the definitions above but it is not complete. That is although types $\tau 2$ and $\tau 3$ are recursive each no object of a given type appears multiple times. Figure 3b shows a similar structure but in this case the 2nd and 3rd nodes in the list are statically recursive. Since the type $\tau 2$ appears multiple times (in node 2 and 3) these two nodes form a complete structure thus we want to replace this set of nodes with a single summary node.

To distinguish between these two cases we perform a connectivity aware detection of the recursive structures which takes connectivity and multiplicity into account ensuring that we only consider two nodes as being recursive if they are part of a *complete recursive structure*. This ensures only nodes that are in repeating and uninteresting parts of a recursive data structure are grouped together.

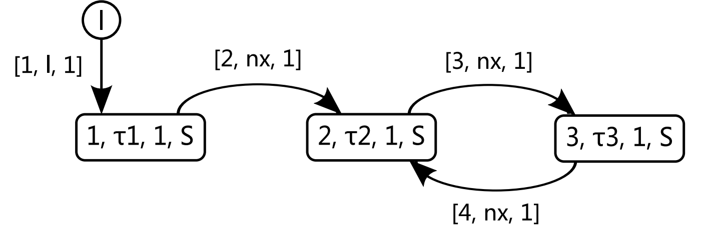


Figure 4: Recursive Cycle

DEFINITION 4 (Complete Recursive Structure). *Two nodes n, n' are part of a* complete recursive structure *if:*

$\exists$ edge e from n to n', $\exists n_\tau$ and a path from n' to n_τ s.t. none of n, n', n_τ or the nodes on the path are safe, *and $n.\text{type} \cap n_\tau.\text{type} \neq \emptyset$.*

3.4 Recursive vs. Back Pointers.

Many programs use back pointers causing the above definition to identify any cyclic structure as recursive, since trivially every node can reach itself and thus every type appears multiple times. This causes the grouping of cycles in the graph into single nodes with the *layout (C)ycle*, which can lead to substantial imprecision. Figure 4 shows an example of such a heap. We can see that even though the abstract heap structure is finite, the back edge will cause our recursive component definition to group the 2nd and 3rd nodes into the same recursive component. To address this and similar problems that arise when distinguishing between bounded and unbounded structures when cyclic structures are present, we modify the recursive definition to ignore back edges.

3.5 Recursive Node Definition

Given the above scenarios and the proposed solutions for handling them we get the following final definition for determining if two nodes are recursive (that is they represent part of the same potentially recursive data structure on the heap).

DEFINITION 5 (Recursive Nodes). *Given the function* depth *which returns the depth of a node in the abstract heap graph, nodes n, n' (where $n \neq n'$) are* recursive *if:*

$\exists$ edge e from n to n', neither of n, n' are safe *and $\exists n_\tau$ s.t. there is a (possibly empty) path $\psi_r = \langle (n^s_1, n^e_1) \ldots (n^s_k, n^e_k) \rangle$ from n' to n_τ s.t. $\forall (n^s_i, n^e_i) \in \psi_r, \text{depth}(n^s_i) < \text{depth}(n^e_i)$ (where* depth *is the depth of the node in the graph), $\forall (n^s_i, n^e_i)$, neither n^s_i or n^e_i is safe and, $n.\text{type} \cap n_\tau.\text{type} \neq \emptyset$.*

4. Composite Components and Array/Collection Grouping

The second contribution of this paper is a method to identify composite structures and equivalence classes of the objects stored in arrays or collections, which has not been studied as extensively as the problem of identifying recursive structures. The approach presented in this paper is based on the definition of a parametric predicate for determining if two nodes represent *equivalent* regions of the heap. The method presented in this section is based on the identification of heap regions based on connectivity information (and is sufficient for most optimization applications) as well as a parametric component which allows for the predicate to be tailored to support other applications as well (for example if we are using a numeric domain we can extend it to keep objects in an array with non-zero values in a given field distinct from objects that must have a zero in this field).

We introduce a notion of *equivalence* of two nodes that captures our intuition of when two nodes n, n' abstract similar regions of the concrete heap. Since the *equivalence* predicate is used to determine the maximum number of out edges each node may have, we can improve efficiency by minimizing the number of equivalence classes created by this relation. The tradeoff between precision and performance that we have found to be acceptable is determined by the following conditions: (1) are all the types represented by the nodes non-recursive (or may both nodes represent recursive types) and (2) what variables can access the objects in the regions abstracted by the nodes?

4.1 Recursive Similarity

Two nodes are *recursive similar* if they both abstract all non-recursive types or they both may abstract objects with recursive types. An example of why this is important is the common construction of k-ary trees using arrays/collections to hold either a recursive subtree or a non-recursive (with respect to the internal tree) leaf object.

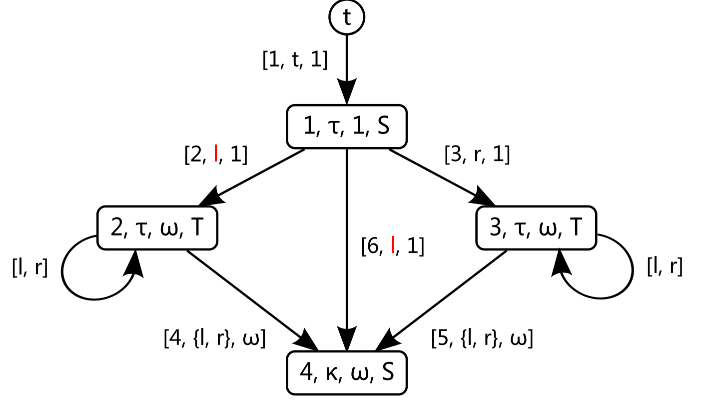
DEFINITION 6 (Recursive Similarity). *Given nodes n, n' and the* statically recursive *type information, n, n' are* recursive similar *iff either of the following holds:*

1. $\exists \tau \in n$.type$, \tau' \in n'$.type *s.t.* τ, τ' *are statically recursive.*
2. *(* $\nexists \tau \in n$.type$, \tau$ *is statically recursive)* $\wedge$ *(* $\nexists \tau' \in n'$.type$, \tau'$ *is statically recursive)*

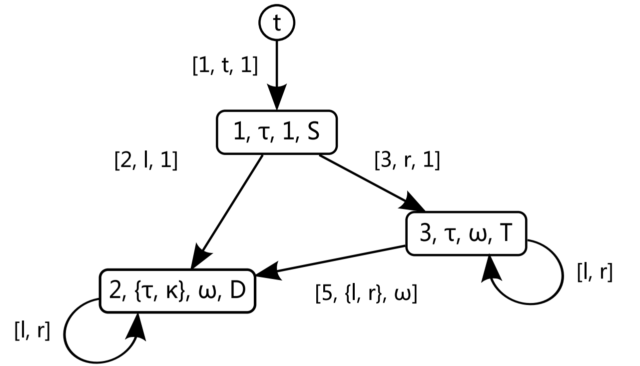
An example of where this heuristic applies is shown in Figure 5a. In this figure we have two types of objects τ, κ which both inherit from the superclass μ (a common way to build a tree structure in Object-Oriented Programing). The class τ is specialized to represent the internal tree structure (via the fields `l` and `r`) which point to objects of type μ. The class κ is the non-recursive leaf class which contains some value and may be referred to by multiple τ tree nodes.

In this case we want to make sure that not only do we distinguish the root node of the tree as well as the left and right sub-trees (which are preserved by the recursive structure identification heuristics in Section 3) but we also want to make sure that the analysis keeps the objects representing the internal tree structure in a disjoint region from the objects representing the leaf objects. Otherwise we would end up merging nodes 2 and 4, as they are both pointed to by an edge with *offset* `l` (highlighted in red if color is available) that starts at node 1 (Def. 8). This would result in a *DAG* region in node 2 and a loss of the overall tree structure as shown in Figure 5b.

At the other end of the range of possible similarity relations, if we were to ensure that regions with differing types were always kept separate the analysis would build unacceptably large tree



(a) Internal Tree and Leaves In Disjoint Regions



(b) Without Use of Recursive Similarity

Figure 5: Recursive Similarity

structures for many programs. For example a compiler may have a large number of classes that inherit from an `Expression` base class which appear in the parse tree structure and are treated uniformly by the program. If we maintained an abstract graph node for each of these types the tree would have a very large branching factor (and potentially depth) causing substantial performance degradation in the analysis.

4.2 Reference Similarity

If we have two nodes n, n' and the objects abstracted in the region by n are all stored in an array A and all the objects in the region abstracted by n' are stored in array A and a second array B then it is reasonable to assume that the programmer has partitioned these objects differently for some reason. Thus, we want to preserve this information by keeping the nodes distinct, we show this situation in Figure 6. We can ensure that the information on which collections and variables refer to which sets of objects is maintained by using the following definition of *reference similarity*.

DEFINITION 7 (Reference Similarity). *We say two nodes n, n' are reference similar if given the set of in edges to n, $E_{\text{in}} = \{e^n_1 \ldots e^n_k\}$, the set of in edges to n', $E'_{\text{in}} = \{e^{n'}_1 \ldots e^{n'}_k\}$, and the set of variables that can reach node n, $V_r = \{v^n_1 \ldots v^n_r\}$, the set of variables that can reach node n', $V'_r = \{v^{n'}_1 \ldots v^{n'}_s\}$, the following holds:*

$$(\{e.\text{offset} \mid e \in E_{\text{in}}\} = \{e'.\text{offset} \mid e' \in E'_{\text{in}}\}) \wedge (V_r = V'_r)$$

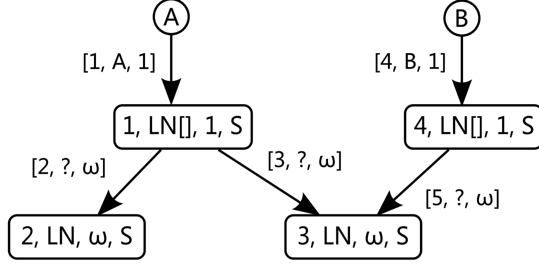


Figure 6: Nodes 2, 3 Not Reference Similar (based on variable reachability)

This definition ensures that if two nodes are treated differently with respect to the types of objects they are stored in or the variables that reach them then they are kept separate. In Figure 6 nodes 2 and 3 are not *reference similar* since node 2 is reachable from variable `A` while node 3 is reachable from both variables `A` and `B`.

4.3 Parametric Node Equivalence

In addition to using the structural information provided by the *recursive similar* and *reference similar* relations we can also provide a parametric component to the grouping operation to support the needs of more specific types of analysis. For example if we are checking a program to ensure that all file reads are exception free we want to distinguish `InputStream` objects that are open from those that are closed even if we have an array of such objects. Similarly if we are interested in checking locking properties we always want to distinguish between objects that are locked and those that are unlocked. Thus our definition allows parametric similarity properties to support specialized analyses that depend on precisely tracking differences of specific properties of interest for the objects in the program.

DEFINITION 8 (Equivalent Nodes/Edges). *Given the above definitions we define* edge equivalence. *Given a node n and two out edges e, e' which start at node n and end at nodes n_e and $n_{e'}$ respectively we say e, e' are* equivalent *if:*

1. e.offset $= e'$.offset
2. $n_e, n_{e'}$ *are* recursive similar
3. $n_e, n_{e'}$ *are* reference similar
4. $n_e, n_{e'}$ *are equivalent for all parametric similarity relations*

5. Region Identification and Grouping

Using the above definitions for identifying recursive structures, composite structures and grouping the contents of collections/arrays we define the method for constructing the logically related regions. Once we have identified a set of nodes that represent a logically related region, based on our region predicates, we need to replace them with a single node that *safely* approximates the properties of the nodes in the set.

5.1 Component Summarization

Before we present the complete region identification/normalization algorithm we describe how the summary nodes are computed. To simplify the computation we perform the summarization in a pairwise manner. When summarizing two nodes, n and n', there are three possibilities. The first is that there are no edges between the nodes, there are only edges in one direction between nodes (from n to n' or n' to n, but not both) and when there are edges from n to n' and from n' to n.

If there are no edges between the nodes we use the *mergeNoEdge* method to compute the summary representation. This method is a simple component-wise operation where the updated *type* label is the union of the two *type* sets, the *linearity* value is ω and the *layout* is the max (the most general) of the two *layout* labels. The case where there are edges from n to n' and from n' to n (*mergeBothWay*) is similar except we always assume the *layout* of the summary node is *(C)ycle* (while this is in general a significant over approximation we have found that the infrequency with which it is used makes this an acceptable definition).

The *mergeOneWay* operation (Algorithm 1) on a pair of nodes that have connecting edges is more complicated. In particular we need to account for the fact that the edge(s) connecting nodes n and n' will affect the *layout* of the new summary node.

Algorithm 1: mergeOneWay

input : graph g, n, n' nodes, *ebt* set of edges from n to n'
n.types $\leftarrow$ n.types $\cup$ n'.types;
n.linearity $\leftarrow$ ω;
n.layout $\leftarrow$ combineLayout(n.layout, n'.layout, *ebt*);
remap all edges incident to n' to be incident to n;
deleteNode(g, n');

The algorithm *combineLayout*(l, l', ebt), is based on a case analysis of the *layout* that results from the possible combinations of the *layouts* for n, n' along with the total number of pointers represented by *ebt* [20]. We enumerate the possible combinations of the *ebt* edges and the *layout* labels and then for each case we use the semantics of the edge and *layout* properties to determine the most general *layout* type that may result from this particular case. For example if we have two *(S)ingleton* nodes connected by an edge of *linearity* 1 then the most general *layout* for a node that summarizes these nodes and the edge is a *(L)ist*.

To merge two arbitrary nodes n, n' we use Algorithm 2 which selects the appropriate method for merging two nodes based on the existence of edges between them.

Algorithm 2: mergeNode

input : node n, n', graph g
if $\exists$ *edges from n to n' and n' to n* **then**
    mergeBothWay(g, n, n');
else if $\exists$ *edges from n to n'* **then**
    mergeOneWay(g, n, n', $\{e \mid e$ from n to $n'\}$);
else if $\exists$ *edges from n' to n* **then**
    mergeOneWay(g, n', n, $\{e \mid e$ from n' to $n\}$);
else
    mergeNoEdge(g, n', n);

5.2 Region Identification/Normalization Algorithm

Once we have the above methods for computing summary nodes for a pair of nodes in the graph we can define the final region identification algorithm. The resulting region grouped model is also a convenient normal form ensuring that the static analysis terminates as the infinite set of *labeled storage shape graphs* is a finite set under the normal form (recursive structures are represented by a bounded number of nodes and each node has a bounded number of out edges, for space we omit a formal proof).

The algorithm is a straightforward iterative identification of pairs of nodes/edges that should be grouped and the replacement of these structures by a summary representation until a fixpoint is reached. After this method terminates the abstract graph model will have all the logically related regions identified and grouped according to the characterizations in Sections 3 and 4.

Algorithm 3: groupRegions

input : graph g
while g *is changing* **do**
  while $\exists$ *node n with edges* e, e' *s.t.* $e \neq e' \wedge e, e'$ *are* equivalent edges **do**
    mergeNode(target of e, target of e', g);
    e.linearity $\leftarrow \omega$;
    deleteEdge(g, e');
  while $\exists$ *nodes* n, n' *that are recursive* **do**
    mergeNode(g, n, n');

6. Case Study and Experimental Evaluation

In this section we look at two case studies that illustrate how the heuristics presented above allow the analysis to group heap objects into regions and how this information can be used to drive a range of memory management optimizations. Both benchmarks are taken from a version of the JOlden [2] suite.

6.1 Em3d

The first program we look at is Em3d which computes electromagnetic field values in a 3–dimensional space by constructing a list of ENode objects, each representing an electric field value and a second list of ENode objects, each of which represents a magnetic field value. To compute how the electric/magnetic field value for a given ENode object is updated at each time step the computeNewValue method uses an array of ENode objects from the opposite field and performs a convolution of these field values and a scaling vector, updating the current field value with the result. The main computation code is shown below:

```
void compute() {
  for(int i = 0; i < this.eNodes.size(); ++i)
    eNodes.get(i).computeNewValue();

  for(int i = 0; i < this.hNodes.size(); ++i)
    hNodes.get(i).computeNewValue();
}

void computeNewValue() {
  for(int i = 0; i < fromCount; i++)
    value -= coeffs[i] * fromNodes[i].value;
}
```

Figure 7 shows the heap structure that is constructed by the program and that is used in the main computation algorithm. To aid clarity we placed dashed lines around the composite structures that represent the magnetic field (in blue if color is available) and the electric field (in green). Variable this points to a single object of type BiGrph, which is the data structure that encapsulates all the objects of interest. The BiGrph object has 2 fields, the hNodes field pointing to a Vector of ENode objects that make up the magnetic field and, the eNodes field pointing to a Vector of ENode objects that make up the electric field. Each of these ENode objects has an array of floats and an array of Enode objects from the opposite field that are used to update the value of the field on each iteration of the field value computation loop. The region analysis identification techniques have precisely grouped all of the heap components in the program into the composite electric/magnetic field structures and even though the overall heap structure is cyclic the analysis has precisely resolved the bipartite graph structure. We note that in this example the definition of *safe nodes* due to non-recursive in edges is critical to ensuring the analysis resolves the heap into a bi-partite structure instead of merging many of the nodes into a single cyclic region.

While the heap is not further modified after construction, and thus there are no opportunities for improved memory collection, the above computation loop is an excellent candidate for altering memory layout to improve spatial locality of the memory accesses. This can be done statically by determining that the lifetimes of the ENode objects are bounded by the lifetime of the Vector they are stored in. Then at allocation time we can co-locate the ENode objects with the Vector [8]. Or we can use this information to provide support for the runtime reallocation of the ENode (and perhaps ENode[] or float[]) objects into contiguous memory pools based on the electric/magnetic structures they are in [12]. Our simple hand implementation of these optimizations on this benchmark resulted in approximately a 7-10% performance improvement, indicating that the information provided by the analysis is able to support sophisticated program transformations resulting in non-trivial performance improvements.

6.2 Barnes-Hut

The bh program performs a gravitational interaction simulation on a set of bodies (the Body objects) using a *fast-multipole* technique with a space decomposition tree. The tree is represented using Cell objects each of which has a Vector containing references to other Cell objects or references to the Body objects. The program also keeps two Vector objects for accessing the bodies, bodyTab and bodyTabRev. The positions (pos), velocities (vel) and acceleration (acc) values of the bodies are represented with composite structures consisting of a MathVector object and a double[].

Using a common OOP idiom the Cell and Body objects both inherit from an abstract Node class. Thus, if we did not use the concept of *recursive similarity* to distinguish between references in the Vector collection to the recursive Cell objects which make up the tree structure and the non-recursive leaf Body objects the analysis would end up grouping the tree and the leaf objects into the same region. However, by distinguishing regions based on their *recursive similarity* the analysis has ensured that the tree structure and the leaf objects are grouped into different regions.

Figure 8 shows the abstract heap model built and used in the stepSystem method of the benchmark (the listing below), where the space decomposition tree is recomputed (the makeTree method), the body-body interactions are computed (the loop with the hackGravity method), and the new acceleration information is propagated (the vprop method).

```
public void stepSystem() {
  root = null;
  makeTree(nstep);

  Iterator<Body> bi = bodyTabRev.iterator();
  while (bi.hasNext())
    bi.next().hackGravity(rsize, root);

  vprop(bodyTabRev, nstep);
}
```

As we can see in Figure 8 the region identification algorithm is able to correctly identify and group all the major components in the overall heap structure. The space decomposition tree is grouped into the region represented by node 17 (although the analysis has overly conservatively assumed the structure may have a *(C)yclic* layout) while the leaf Body objects are represented separately by node 14. The analysis has also grouped the composite MathVector/double[] structures and has maintained the separation of these structures when they abstract distinct structures and are stored in different types or in different fields.

The bh program has many opportunities to apply the optimizations discussed in the introduction. In particular the information computed by the analysis in this paper enables opportunities that could not be previously exploited due to a lack of sufficiently precise region identification.

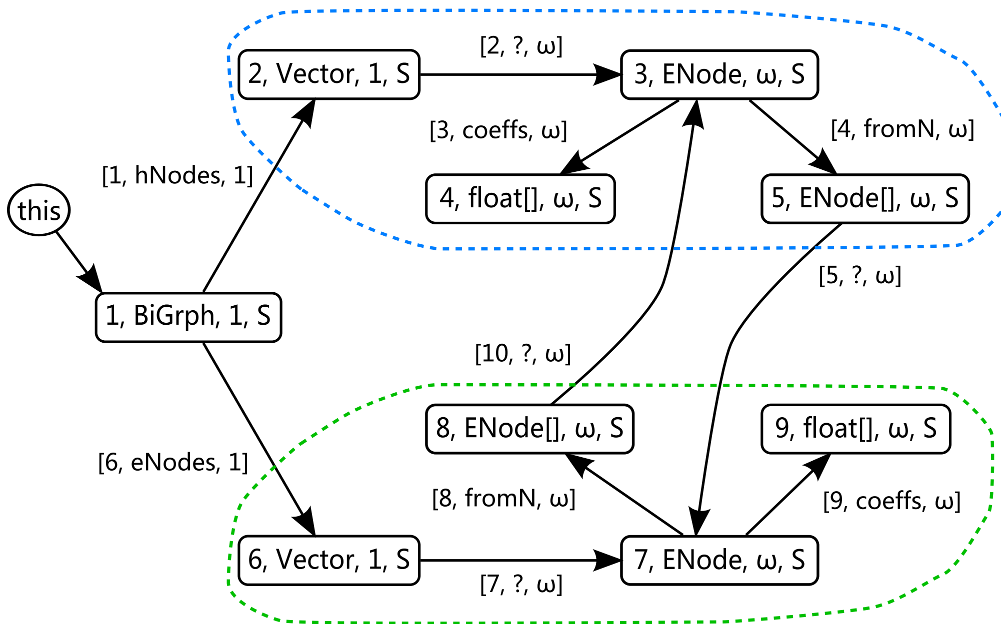


Figure 7: Abstract Heap in Em3d

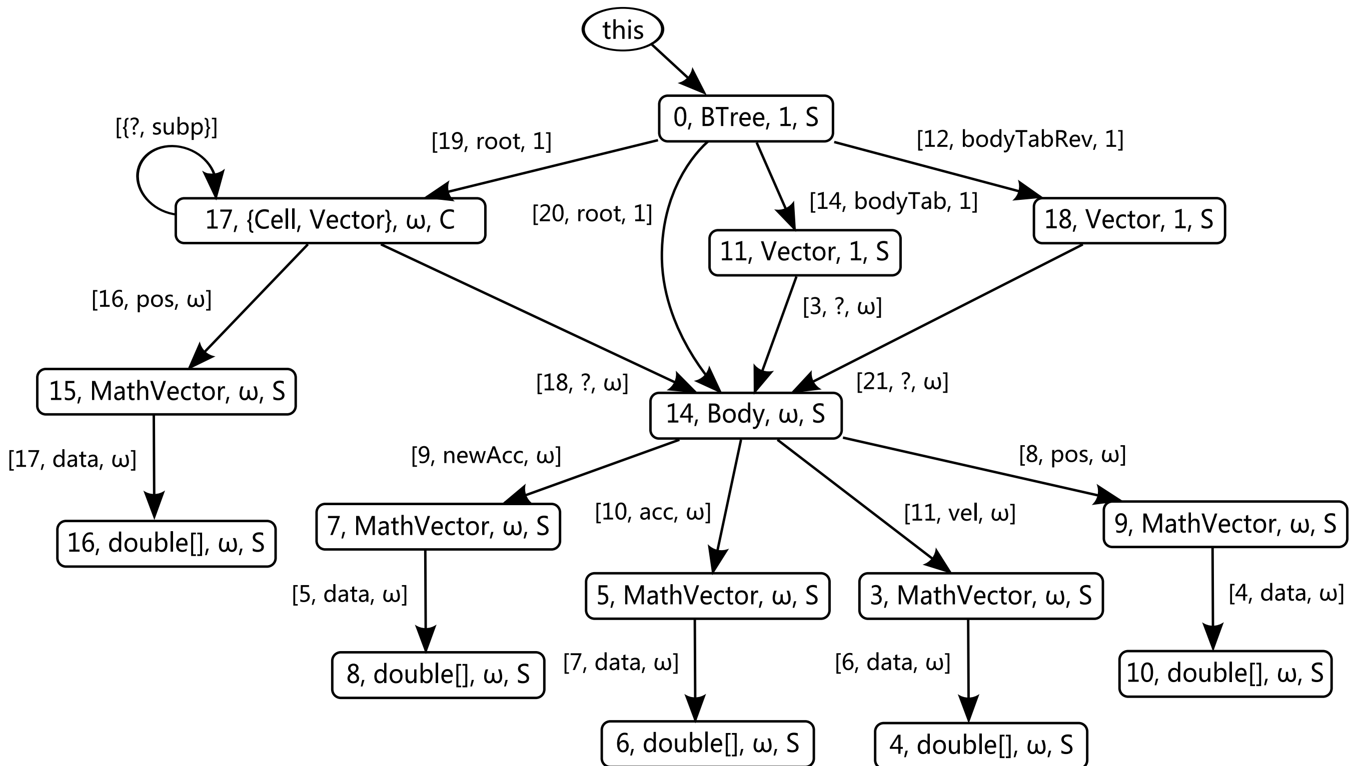


Figure 8: Abstract Heap in bh

The first possible optimization is the use of pool allocation [16] for the space decomposition tree (node 17) which is allocated in the makeTree method and then becomes dead at the root = null assignment on the next loop iteration. By pool allocating this we can collect the entire tree as one block instead of requiring the GC algorithm to traverse and collect each node in the tree one object at a time (reducing the number of objects that the garbage collector needs to reclaim by about 11%) and increasing the spatial locality of the accesses to the tree (which improves the performance of the program by 3-4% percent).

Given the structure of the heap, the two phases of computation and the limited pointer writes during the hackGravity method we can profitably apply parallel and region based collection [15]. This allows us to reduce the GC overhead by collecting dead MathVector objects in the regions for the acc, vel, and pos fields while the newAcc values are being computed in the hackGravity method. Similarly we can collect objects in the space decomposition tree and newAcc field regions while the mutator is in the vprop method. This parallel, region-specific collection greatly reduces the GC pause times while only requiring the collector/mutator to lock once on entry to these methods.

If we include *sharing* information as described in [22] we can determine that the double[] (where the size of the arrays is a small compile time constant) stored in the MathVector objects are never shared between MathVector objects and thus are good candidates for co-location [12, 8]. This has the beneficial effect of increasing the data locality and removing many redundant loads resulting in a 12% reduction in the runtime of the single threaded program, as well as reducing the size of the MathVector/Array composite structure object by a pointer (and the overhead of an array), resulting in a 37% reduction in memory usage.

Finally, if we again use the sharing information in [22] we can statically determine when each of the MathVector/double[] objects becomes dead and can insert explicit collection code for them [13]. This transformation reduces the number of objects that the GC needs to collect by a factor of about 52% (since these objects are immutable there are many of these created for each body object). If we perform this optimization with the pool allocation of the space decomposition tree then all of the objects can be collected statically eliminating the need for the collector entirely.

These transformations allow for the efficient collection (by collecting individual objects or entire pools) of all the dead objects created during this main computation portion and for the location of temporally related objects into contiguous parts of memory. Thus, this benchmark demonstrates how the precision of the region analysis presented in this paper enables the application of a number of powerful program optimizations that reduce the memory requirements, reduce garbage collection costs, and to improve the performance of the program.

6.3 Experimental Evaluation.

We have implemented a shape analyzer based on the region identification methods and instrumentation properties presented in this paper and evaluated the effectiveness and efficiency of the analysis on programs from SPECjvm98 [25] and a version of the JOlden suite. The JOlden suite contains pointer-intensive kernels that make use of recursive procedures, inheritance, and virtual methods. We modified the suite to use modern Java programming idioms. The benchmarks raytrace and db are taken from SPECjvm98.

The analysis algorithm was written in C++ and compiled using MSVC 8.0. The analysis was run on a 2.6 GHz Intel quad-core machine with 4 GB of RAM (although memory consumption never exceeded 120 MB).

For each of the benchmarks we provide a brief description of some of the major structures/features that are in the program.

Benchmark	LOC	Description	Analysis Time
bisort	560	Tree w/ Mod	0.26s
mst	668	Cycle w/ Struct.	0.12s
tsp	910	Tree to Cycle	0.15s
em3d	1103	Bipartite Graph	0.31s
perimeter	1114	Tree w/ Parent Ptr	0.91s
health	1269	Tree w/ Mod	1.25s
voronoi	1324	Cycle w/ Struct	1.80s
power	1752	Lists of Lists	0.36s
bh	2304	N-Body Sim w/ Mod	1.84s
db	1985	Shared/Mod Arrays	1.42s
raytrace	5809	Shared/Cycle/Tree	37.09s

Figure 9: LOC is for the normalized program representation including library stubs required by the analysis. Analysis Time is the analysis time for the analysis in seconds.

We mention the major data structures used (Trees, Lists of Lists, Cycles, etc.) and if the program heavily modifies the data structures (w/ Mod). Some of the benchmarks have slightly more nuanced structures — mst and voronoi which build globally cyclic structures that have significant local structure, bh which has a complex space-decomposition tree and sharing relations, and raytrace which builds a large multi-component structure which has cyclic structures, tree structures, and substantial sharing throughout. We also note that tsp and voronoi begin with tree structures and process them building up a final cyclic structure during the program. These benchmarks thus exercise a wide range of features in the analysis based on the types of structures built, modification of these structures, sharing of the structures, use of multi-component structures, and the use of arrays/collections.[1]

As our interest in this paper is primarily in the development of a heap analysis that can support a range of memory management and optimization techniques rather than in the performance of a specific GC method we focus on the cost of running the analysis to produce the region information. We note that the region information produced for all of the benchmarks is similar in quality to the results in the case studies (thus many of the same optimizations could be applied) and that the runtimes are on the order of seconds even for programs like bh and raytrace which make use of complex data structures, a number of classes from java.util/java.io and have nontrivial amounts of sharing between data structures.

7. Related Work

There has been a significant amount of work on developing static techniques to improve the allocation [16, 4], layout [12] and collection [16, 5, 13, 15] of memory in object oriented programs. These techniques have introduced a variety of methods for computing region information based on static partitions computed using a range of points-to analyses and are capable of scaling to large programs. However, the imprecision of fixed partitioning and flow insensitivity in parts of the analysis limits their ability to precisely analyze many programs that destructively rearrange regions and limits the ability to disambiguate components of larger composite structures (i.e. the 2 distinct regions of ENode objects in the overall cyclic heap structure in Em3d or disambiguating the Body objects from the space decomposition tree in Barnes-Hut). Thus the performance improvement achieved by the optimizations proposed in these papers, while good, is limited by the precision of the analysis results.

Other recent heap analysis work has focused on the precise modeling of destructive updates and their effect on the structure

[1] See www.software.imdea.org/~marron/ for benchmark code, examples of the analysis results, and an executable analysis demo.

of the heap, TVLA [19, 18, 27, 24], separation logic based approaches [1, 28, 11]. While these techniques can model, with a very high degree of precision, many complex heap operations they currently impose limitations that make region analysis infeasible for many programs. In particular the current formulations are restricted to programs that manipulate lists (or trees) and restrict the amount of sharing between regions. As Separation Logic and TVLA are general purpose frameworks/logics the work in these papers could be extended as described in this work. However, to the best of our knowledge this extension has not been done. Thus, many of the benchmarks examined in this paper currently cannot be analyzed with these methods, including bh, em3d, voronoi, and raytrace, all of which have substantial opportunities for the application of various region based optimizations.

8. Conclusion

The analysis presented in this paper presents an important development in applying shape analysis techniques to real world programs as it can precisely and efficiently deal with the types of data structures and programmatic events that occur in realistic programs. In particular the formalization applies to any type of recursive data structures (as opposed to just lists or trees, and it supports composite data structures that have non-trivial sharing between them), it can precisely model many types of structures which are merged in simpler points-to style approaches, and it supports more precise grouping of the contents of collections (arrays or collections from `java.util`) than is possible with other methods.

Our experiments demonstrate that the proposed region identification method can be used to precisely and efficiently identify and group logically related regions of the heap (recursive data structures, composite structures composed of multiple objects and the contents of arrays/collections). Further our case studies demonstrate that the results of the analysis can be effectively used to support memory allocation/layout/collection optimization applications. Based on these results we believe that the proposed approach presents a basis for a heap analysis that can be used in practice to provide detailed heap information for a range of optimization applications that rely on region information and we are currently working on improving the practicality of the analysis by developing on techniques to scale it to larger programs.

Acknowledgements. We would like to thank the reviewers for their comments and suggestions. This work was funded in part by EU projects FET *HATS* and 06042-ESPASS, Spanish Ministry of Science and Industry projects TIN-2008-05624 *DOVES* and FIT-340005-2007-14, and CAM project S-0505/TIC/0407 *PROMESAS*. This work was also funded via NSF awards CCF-0541315, CNS-0831462, and CCF-0540600.

References

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[2] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.

[3] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[4] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM*, 2004.

[5] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *ISMM*, 2006.

[6] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *PLDI*, 1994.

[8] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI*, 2000.

[9] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[10] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.

[11] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.

[12] S. Z. Guyer and K. S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *OOPSLA*, 2004.

[13] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *PLDI*, 2006.

[14] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.

[15] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *OOPSLA*, 2003.

[16] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.

[17] C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.

[18] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.

[19] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In R. Cousot, editor, *VMCAI*, 2005.

[20] M. Marron. Modeling the heap: A practical approach. Phd. thesis, University of New Mexico, 2008.

[21] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.

[22] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.

[23] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.

[24] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[25] Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[27] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.

[28] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *CAV*, 2008.