The Runtime Structure of Object Ownership

Nick Mitchell

IBM TJ Watson Research Center
19 Skyline Drive, Hawthorne NY 10532
nickm@us.ibm.com

Abstract. Object-oriented programs often require large heaps to run properly or meet performance goals. They use high-overhead collections, bulky data models, and large caches. Discovering this is quite challenging. Manual browsing and flat summaries do not scale to complex graphs with 20 million objects. Context is crucial to understanding responsibility and inefficient object connectivity.

We summarize memory footprint with help from the dominator relation. Each dominator tree captures unique ownership. Edges between trees capture responsibility. We introduce a set of *ownership structures*, and quantify their abundance. We aggregate these structures, and use thresholds to identify important aggregates. We introduce the *ownership graph* to summarize responsibility, and *backbone equivalence* to aggregate patterns within trees. Our implementation quickly generates concise summaries. In two minutes, it generates a 14-node ownership graph from 29 million objects. Backbone equivalence identifies a handful of patterns that account for 80% of a tree's footprint.

1 Introduction

In this paper, we consider the problem excessive memory footprint in object-oriented programs: for certain intervals of time, the live objects exceed available or desired memory bounds. Excessive memory footprint has many root causes. Some data structures impose a high per-element overhead, such as hash sets with explicit chaining, or tree maps. Data models often include duplicate or unnecessary fields, or extend modeling frameworks with a high base-class memory cost. There may be objects that, while no longer needed, remain live [34, 39], such as when the memory for an Eclipse [17] plugin persists beyond its last use. Often, to mask unresolved performance problems, applications aggressively cache data (using inefficient data structures and bulky data models).

To isolate the root causes for large object graph size requires understanding both responsibility and internal content: the program may hold on to objects longer than expected, or may use data structures built up in inefficient ways. We analyze this combination of *ownership structures* by summarizing the state of the heap — at any moment in time within the interval of excessive footprint. In contrast, techniques such as heap [34, 35, 40, 43], space [36], shape [32], lexical [6], or cost-center [37] profiling collect aggregate summaries of allocation sites. Profiling dramatically slows down the program, gives no information about responsibility

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 74–98, 2006.

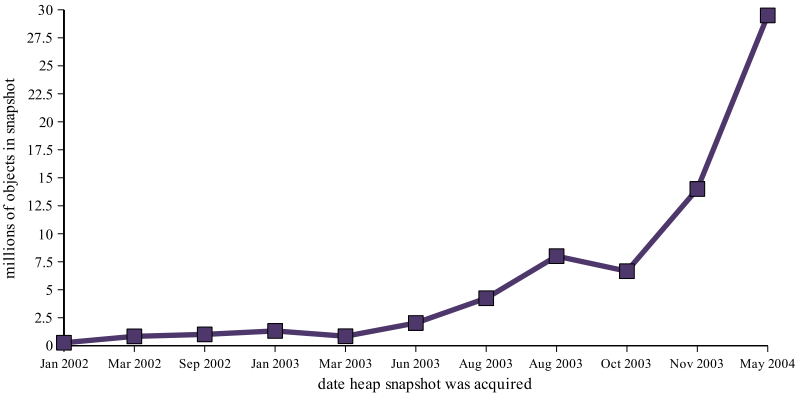


Fig. 1. Growth in the size of Java heaps in recent years

Table 1. A commonly used, but not especially useful, graph summary: aggregate objects by data type, and then apply a threshold to show only the top few

type	objects	bytes
`primitive arrays`	3,657,979	223,858,288
`java/lang/String`	2,500,389	80,012,448
`java/util/HashMap$Entry`	2,307,577	73,842,464
`java/util/HashMap$Entry[]`	220,683	57,726,696
`customer data type`	338,601	48,758,544
`java/lang/Object[]`	506,735	24,721,536

or internal content, and conflates the problem of excessive temporary creation with the problem of excessive memory footprint.

The task of summarizing the state of the heap [12, 20, 9, 29, 19, 32] at any moment in time [3, 31, 18, 25] is one of graph summarization. In this case, the graph's nodes are objects, and the edges are references between them. Summarizing the responsibility and internal content of these graphs is, from our experience with dozens of large-scale object-oriented programs, quite challenging. In part, this is because these object graphs are very large. In Figure 1, we show typical object graph sizes from a variety of large-scale applications. Over the years, this figure shows that the problem has grown worse. Contemporary server applications commonly have tens of millions of live objects at any moment in time.

Furthermore, the way objects are linked together defeats easy summarization. A good summary would identify a small number of features that account for much of the graph's size. Achieving this 80/20 point, especially for large, complex graphs, is challenging. Many commercial memory analysis tools [3, 31, 18] *aggregate* by data type, and then chooses a *threshold* to show those biggest types. This technique produces a table such as Table 1. Typically, generic data types float to the top. Even for the customer-specific types, the table gives us no sense of who is responsible, or how the instances are structured; e.g. are the instance of these types part of a single large collection, or several smaller ones? These same

tools also provide *filters* to ignore third-party providers such as J2SE, AWT, and database code. But, as the table shows, those third parties often form the bulk of the graph's size. In addition, they often provide the collections that (perhaps inefficiently) glue together the graph's objects.

Filters, aggregations, and thresholds are essential elements of summarization, but must be applied carefully. Thresholds help to focus on the biggest contributors, but the biggest contributors are not single data types, or even single locations in the graph. As we will show in Section 4, those 3.6 million primitive arrays in Table 1 are placed in many distinct locations. Thus, at first sight, they appear to be scattered throughout the (18-million node) graph. However, we will show that only two distinct *patterns* of locations that account for 80% of the largest data structure's size. This careful combination of aggregation and thresholding can produce concise summaries of internal content.

The same care is necessary when summarizing responsibility. If the object graph is *tree-like* (i.e. a diamond-free flow graph), the problem of summarizing ownership structure reduces to that of summarizing content; responsibility is clear when each object has a single owner. As we will quantify later, object graphs are not at all tree-like. For example, in many cases two unrelated program mechanisms share responsibility for the objects in a data structure; in turn, those two mechanisms themselves are shared by higher-level mechanisms. The result is a vast web of responsibility. We can not arbitrarily filter out edges that result in sharing, even if it does have the desirable property of reducing the responsibility structure to a tree [23].

This paper has four main contributions.

Analysis Methodology. We decompose the analysis of ownership structures into two subproblems, by leveraging the dominator forest [22, 14, 33] of the graph. We use the dominator relation for two reasons. First, it identifies the maximum *unique* ownership within the graph. This aligns well with our distinction between responsibility and content. The edges between trees in this forest capture responsibility, and the elements of a dominator tree capture content. Second, a graph has a single, well-defined dominator forest; a depth-first traversal, in contrast, also produces a spanning tree, but one that depends on an arbitrary ordering of graph roots.

Catalog of Ownership Structures. We develop a catalog of ownership structures, for both responsibility and for internal content. For example, for content we introduce six categories of *backbones*, those structures that allow collections to grow or shrink. We justify their importance by quantifying their prevalence in large-scale applications; being common, they will serve as powerful units of aggregation and filtering. In addition, we demonstrate that categorizing content by backbone structure provides a powerful, if flat, summary of content.

Algorithm for Summarizing Responsibility in Graphs. Beyond flat summaries, we provide summarization algorithms that use this catalog of structures. The summary of responsibility is an *ownership graph*, itself a graph, where each

node is an aggregation of ownership structures. We show how the dominator relation alone is a powerful tool for summarizing responsibility; e.g. in one server application, it reduces 29 million nodes to 191 thousand dominator trees (a 99% reduction). We also show how six other structures of responsibility allow us to reduce that summary to a 14-node ownership graph. Our implementation generates that summary automatically in around two minutes.

Algorithm for Summarizing Content in Trees. We summarize the content of a tree by aggregating according to *backbone equivalence*. We introduce two equivalence relations that group together the nodes that may be in widely divergent tree locations, but should be considered as part of a single unit. For example, in a hash set that contains hash sets of strings, there may be millions of strings. All of the strings are backbone-equivalent. In Section 4.3, we demonstrate that this enables a form of analysis that identifies the largest patterns in the largest trees. For example, we show how to locate the set of distinct patterns within a tree in which a dominant data type (such as those shown in Table 1) occur. We demonstrate that a handful of patterns account for most of a hot type's footprint, despite it being in millions of distinct locations in the tree.

Section 3 covers the catalog and algorithms for responsibility, and Section 4 covers the issues of content. We begin with a short discussion of the input to our analysis: seven snapshots from large-scale applications, and seven benchmarks.

2 Object Reference Graphs

To diagnose a memory footprint problem, we analyze a snapshot of its live objects and the references between them.[1] We treat a snapshot as a directed graph, commonly termed an *object reference graph*. The nodes of this graph represent objects and the edges represent a field of one object referring to another. In addition to objects and references, we assume only that the snapshot associates a data type and an instance size with each object. Typically, object reference graphs are neither connected, nor flow graphs (in the sense of [22], where the graph is rooted); we will see more detail and quantifications in Section 3.

Table 2 introduces the applications and SPEC JVM98 benchmarks [41] we study in this paper. Real applications frequently have ten or even twenty million live objects; for this paper, we decided to present a spectrum of graph sizes from real applications. Notice that even `A2'` is large; it represents a web application server just after server startup has completed. For all fourteen snapshots, the numbers reflect only live objects. We use the JVM's built-in support for generating snapshots, which halts all threads, forces a garbage collection, then writes the snapshot to disk. In the case of the benchmarks, we use the maximally-sized run, and take several dozen snapshots over the course of each benchmark's run. We document the largest of those snapshots.

[1] To manually trigger a heap snapshot with the IBM JVM, send a SIGQUIT signal to the JVM process. In the rare case of a short spike in memory footprint, set the heap size so as to cause an out of memory exception upon the spike. At this point, the JVM automatically triggers a heap snapshot.

Table 2. The heap snapshots we analyze in this paper. They include both real applications and benchmarks, divided into the top and bottom half of this table.

application	objects	bytes	description
A1	29,181,452	1,433,937,576	telecom transaction server
A2	20,952,120	1,041,348,768	multi-user collaboration server
A3	17,871,063	818,425,984	e-commerce transaction server
A2'	4,391,183	241,631,368	A2, just after server startup
A4	4,289,704	232,742,536	catalog management server
A5	4,099,743	269,782,704	rich client
A6	3,757,828	99,909,500	an Eclipse-based rich client
mtrt	509,170	13,590,874	
db	342,725	10,569,334	
javac	316,857	10,593,467	
jess	83,815	9,827,946	SPECjvm98 benchmarks
jack	37,949	4,193,140	
mpegaudio	9,997	947,196	
compress	7,696	808,741	

3 Summarizing Responsibility Within Graphs

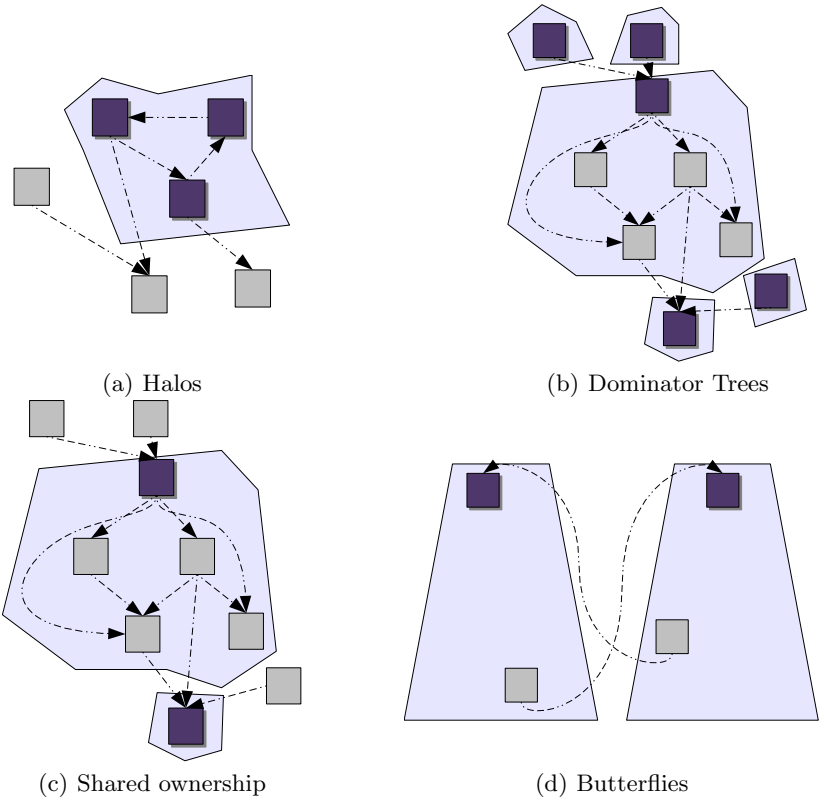
This section introduces a way to summarize the responsibility for the memory footprint of an object reference graph. We first introduce important structural and semantic *graph* properties, and quantify the extent to which these properties occur in both real applications and benchmarks. We then present an algorithm to compute an *ownership graph*, a new graph that succinctly summarizes the ownership structures within a given graph.

3.1 Four Common Graph Structures

We identify four common graph properties of subgraphs within an object reference graph. They do not depend on features of the language, runtime environment, or application. Figure 2 illustrates these four purely structural graph properties: halos, unique ownership, shared ownership, and butterflies.

Halos. Many object reference graphs include structures such as illustrated in Figure 2(a). This graph has two roots, one of which is a proper graph root (a node with no parents). The three objects that form a cycle combine to make up the second root. We term this cycle at the top of the graph a "halo". A halo is a strongly-connected component in which no constituent has a parent outside of the component's subgraph.[2]

[2] Sometimes, the objects in a halo are garbage; e.g. HPROF [43] does not collect garbage prior to writing a heap snapshot to disk. More often, non-Java mechanisms reference members of a halo, but the snapshot does not report them; e.g. if the garbage collector is not type accurate, this information may not be available.



(a) Halos (b) Dominator Trees

(c) Shared ownership (d) Butterflies

Fig. 2. Four common structural properties of graphs

Dominator Trees. The dominator relation [22] applied to graphs of memory describes the unique ownership of objects [7]. A relatively small set of nodes often dominate large subsets of the graph. The immediate dominator relation imposes a spanning forest over the graph. Figure 2(b) illustrates a graph whose dominator forest consists of five trees: four single-node trees and one five-node tree. We highlight the root of each dominator tree with a darker shade.

Shared Ownership. For those nodes that are roots of the dominator forest, but not roots of the graph, the ownership responsibility is shared. Figure 2(c) highlights the two dominator trees of Figure 2(b) with shared ownership. Table 3 shows how, among a number of real applications, more than 75% of the dominator trees have shared ownership; we discuss this table in more detail below.

Butterflies. Mutually shared ownership arises when one node of a dominator tree points to the root of another dominator tree, while a node of that other dominator tree points back to the root of the first tree. Figure 2(d) illustrates a case where two dominator trees mutually own each other; we refer to these structures as "butterflies". These structures are common in real applications, where 7–54% of dominator trees are involved in butterflies.

Table 3. The structural properties of graphs; the fifth and sixth columns show the fraction of the dominator trees that are shared and involved in butterflies

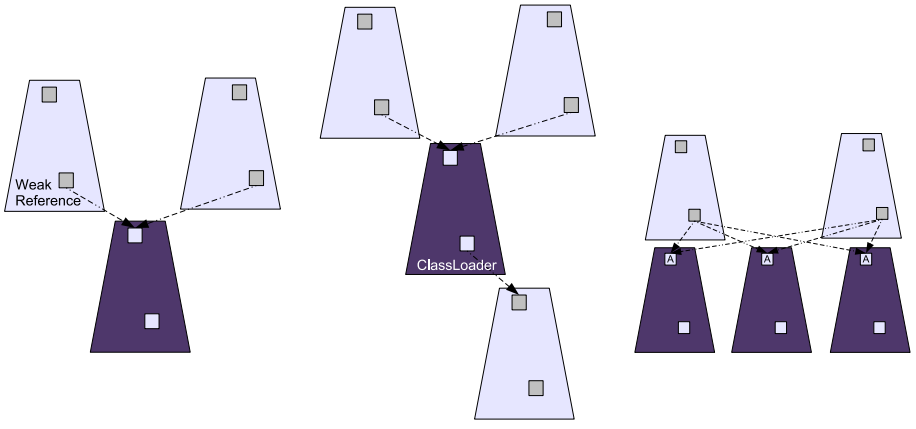
application	halos	avg. objects per domtree	shared domtrees	domtrees in a butterfly
A1	152	153	81%	25%
A2	3,496	41	91%	24%
A3	1,064	310	87%	54%
A2'	1,776	39	76%	9%
A4	3,828	27	78%	42%
A5	3,492	43	67%	7%
A6	826	103	72%	13%
mtrt	25	3	2%	<1%
db	7	5	32%	<1%
javac	27	8	49%	16%
jess	8	3	6%	<1%
jack	27	3	24%	<1%
mpegaudio	117	8	40%	<1%
compress	26	6	45%	<1%

Table 3 summarizes the structural properties for the applications and benchmarks of Table 2. The real applications have many halos, large dominator trees, and a great deal of shared and butterfly ownership. Only one benchmark, `javac`, exhibits characteristics somewhat like the real applications.

3.2 Three Structures Derived from the Language and Data Types

We supplement structural information with three pieces of semantic information: contingent ownership, the class loader frontier, and context-identical dominator trees, as illustrated in Figure 3. The first two draw upon language features, and the third takes object data types into account.

Contingent Ownership. Some language mechanisms reference objects, but do not impact their lifetime. We choose to filter out these references, for the purposes of summarizing the responsibility for graph nodes. Java applications commonly use two such mechanisms: weak references, and the finalizer queue. The constructor of a `WeakReference` creates a new object that references a provided object; the garbage collector ignores this reference when determining liveness. For example, in the situation illustrated by Figure 3(a), one of the two referents to the bottom dominator tree is from a weak reference. From structural perspective, the bottom tree has shared ownership; but it is more natural to consider the weak ownership to be *contingent* upon the persistence of the strong reference. Similarly, instances of a class with a `finalize` method will be referenced by the finalizer queue; but, again, these references do not impact liveness. In addition, we choose to filter references due to Java soft references. These references informs the garbage collector that, in the absence of other strong references, to free the object when memory becomes tight.



(a) Contingent ownership (b) Class loader frontier (c) Context-identical trees

Fig. 3. Three common semantic properties of graphs

Definition 1 (Contingent Ownership). *We say that an edge* (n', n) *offers* contingent ownership *to* n *if* n' *is weak, soft, or part of the finalizer queue, and there exists at least one other edge* (n'', n) *such that* n'' *is not weak, soft, or part of the finalizer queue. We say that a reference* (n, n') *offers* strong *contingent ownership if there is exactly one such* n''*.*

Table 4 shows the fraction of shared dominator trees that have this property. The real applications all have thousands of dominator trees with contingent ownership, and on average 52% of the contingent ownership is strong. The benchmarks have a higher proportion of strong contingent ownership: 78%.

Class Loader Frontier. Real applications have a large boundary between dominator trees headed by class loader mechanisms and trees of non-class loading mechanisms. Figure 3(b) illustrates a case with four dominator trees located on either side of this boundary. This boundary is large because real applications make heavy use of class loaders, and they commonly have shared ownership. Table 5 shows that real applications have as many as 38 thousand dominator trees headed by class loader data types; on average, 29% of the class loader dominator trees from these seven snapshots were shared. Further complicating matters, these shared class loader dominator trees tend to reach nearly all objects. This is because, in real applications, the class objects very often reach a substantial portion of the graph. Next, the class loader dominator trees are usually reachable from a wide assortment of application, framework, and JVM mechanisms. For example, to isolate plugins, the Eclipse IDE uses a separate class loader for each plugin; its plugin system reaches the class loader mechanism, which in turn reaches many of the objects. The result is a highly tangled web of edges that connect the class loader and other trees.

We say that dominator trees that are headed by an instance of a class loader data type, and that are on either side of the boundary between class loader

Table 4. The number of dominator trees that are contingently owned, and strongly so, compared to the total number of shared dominator trees

application	shared domtrees	contingently owned	strongly contingent
A1	155,069	2,630	1,235
A2	472,177	5,324	1,943
A3	502,534	3,964	2,331
A2'	85,100	3,851	1,624
A4	121,623	33,208	29,984
A5	121,623	3,502	785
A6	26,430	733	294
mtrt	2,545	45	34
db	20,795	26	22
javac	19,830	1,514	1,503
jess	1,796	24	20
jack	3,103	113	100
mpegaudio	506	79	19
compress	542	116	105

mechanisms and all others are said to be on the *class loader frontier*. The fourth column of Table 5 shows the number of dominator trees that lie on this frontier. All of the benchmarks have a small, and roughly equal number of shared class loader dominator trees that are on this frontier; this, despite a widely varying range of shared dominator trees across the benchmarks (as shown in the second column of Table 4). The real applications have a varied, and much larger, class loader frontier. This reflects a richer usage of the class loader mechanism, compared to the benchmarks.

Table 5. The number of dominator trees headed by class loader mechanisms, the number of those that have shared ownership, and the number of dominator trees that are on the class loader frontier

application	class loader total	class loader shared	class loader frontier
A1	8,297	1,032	4,550
A2	26,676	1,008	3,030
A3	38,395	133	3,768
A2'	19,080	959	2,449
A4	5,475	396	1,127
A5	5,410	363	1,259
A6	1,017	120	522
mtrt	51	8	29
db	46	8	21
javac	48	8	22
jess	135	8	23
jack	48	8	29
mpegaudio	47	8	23
compress	47	8	22

Context Equality. Often, a large number of non-contingently owned dominator trees are headed by nodes of the same type and have identical ownership. Figure 3(c) illustrates a case of three context-identical dominator trees: all three are headed by nodes of type `A`, and the set of dominator trees to which their predecessors belong is the same. For example, in a server application, this kind of structure occurs with the per-user session data. The session data structures are often simultaneously stored in two collections, under two different roots. Hence, each is shared, but in the same way. In another common situation, an application allocates and manages Java data structures outside of Java. All that is visible from a Java heap snapshot are many of those data structures with no visible Java roots: the same type of data structures, all in the same (in this case, empty) context. We can leverage this kind of similarity.

Definition 2 (Context-identical). *Let n be a node in a graph, $R(n)$ be the root node of the dominator tree in which that node belongs, $P(n)$ be the set of predecessor nodes of n that do not have contingent ownership over n, and $T(n)$ be the type of a node n. Let $I(n) = \{T(R(p)) : p \in P(R(n))\}$, i.e. the types of the root nodes of the predecessors of n's dominator tree root. We say two nodes n_1 and n_2 are part of* context identical *dominator trees if $T(R(n_1)) = T(R(n_2))$ and $I(n_1) = I(n_2)$.*

Under this definition of equality, we can group dominator trees into equivalence classes. Table 6 shows the number and average size of context-identical equivalence classes for our suite of applications and benchmarks. In real applications, there are typically many thousands of such classes, with a dozen or so dominator trees per class.

Table 6. The number and average size of the context-identical equivalence classes from a variety of applications and benchmarks

application	context-identical equiv. classes	avg. domtrees per equiv. class
`A1`	4,420	13
`A2`	32,087	6
`A3`	36,464	9
`A2'`	3,190	10
`A4`	1,837	31
`A5`	2,078	15
`A6`	2,011	5
`mtrt`	140	11
`db`	9	1,706
`javac`	438	16
`jess`	72	7
`jack`	71	14
`mpegaudio`	8	3
`compress`	5	21

3.3 The Ownership Graph

We demonstrate an algorithm that, given an object reference graph, produces a new *ownership graph* that concisely summarizes responsibility within the input graph. To compute the ownership graph, the algorithm performs a chain of *graph edits*, each of which filters, aggregates, or applies thresholds to the nodes in an object reference graph.

Definition 3 (Graph Edit). *Given a graph G, a* graph edit *E_G is (C, D_n, D_e); C is the* collapsing relation*, an $N : 1$ relation among the nodes of G; D_n and D_e are, respectively, the node and edge* delete sets*, and are subsets of the nodes and edges of G, respectively. We term the range of the collapsing relation as the set of* canonical *nodes of the edit. The* deleted graph *is the subgraph of G consisting of edges either in D_e or whose target is in D_n; its nodes are the nodes of G.*

Applying a graph edit yields a new, reduced, graph that preserves the reachability of the input graph. When applying a chain of graph edits, each edit takes as input the reduced graph generated by the previous graph edit.

Definition 4 (Reduced Graph). *Given a graph edit E_G, define the* reduced graph *of E_G to be the graph R whose nodes are the canonical nodes of E_G and whose edges are the union of edges from G renamed according to the collapsing relation, and edges from the transitive closure of the deleted graph of E_G.*

Each node in a reduced graph represents an aggregation of nodes from previous graphs. Since each collapsing relation is a tree relation (i.e. it is $N : 1$ from nodes to nodes of the input graph), the correspondence between a node of a reduced graph to the nodes of any previous reduced graph is just the transitive closure of the inverse of the collapsing relations.

Definition 5 (Contained Nodes). *Let R be a reduced graph derived, via a chain of graph edits, from a graph G. Define the* contained node *set of $r \in R$ relative to G to be the set of $g \in G$ encountered on a traversal, from r, of the composition of the inverse of the collapsing relations of the chain of graph edits that led to R.*

Using a combination of five kinds graph edits, some applied multiple times, we construct concise ownership graphs. We now define those five kinds of edits, and subsequently describe an ownership graph construction algorithm.

Dominator Edit. Compute a representative for each halo of the input graph; we find the set of representatives that, on any depth-first traversal of the input, have only back edges incoming. The union of this set of halo representatives with those nodes that have no incoming edges form the *root set* of the graph. Given this root set, compute the dominator forest of the input graph.[3] From

[3] The dominator algorithm we use [22] assumes that the input is a flow graph. In our case, we use an implicit start vertex: one that points to the computed root set.

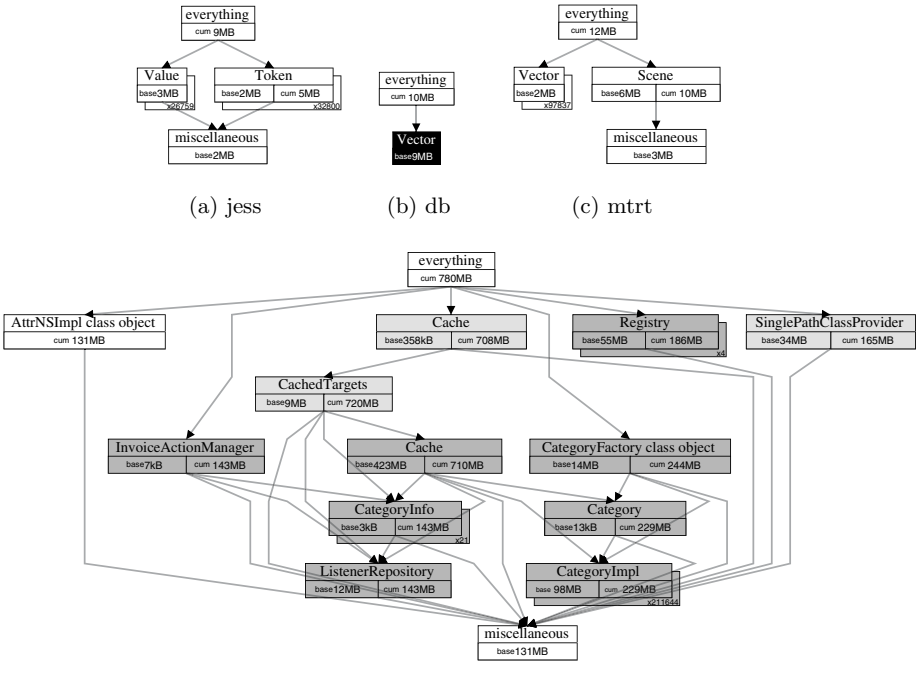
this forest, we define a graph edit (C, D_e, D_n). The collapsing relation C maps a node to its dominator forest root; the deleted edge set D_e consists of edges that cross the class loader frontier or that have only contingent ownership; the deleted node set D_n is empty. This edit collapses the dominator trees into single nodes. It will also remove the shared ownership from dominator trees that are strongly contingently owned.

Context-identical Edit. For each node n of the input graph, compute a representative type T_n. In the case where each node is a dominator tree, we choose this representative type to be the node type of the head of the dominator tree; this will not necessarily be the case when this graph edit is applied subsequent to graph edits other than the `dominator` edit. In the case where each node is a collection of dominator trees whose heads are of uniform type, we choose the representative type to be that type. Otherwise, we say the representative type is undefined. Let the parent set of a node n, P_n, be the set of predecessor nodes of n. Group the nodes of the input graph according to equality, for each graph node n, of the pair (P_n, T_n). For the remaining equivalence classes, choose an arbitrary representative node. The context-identical collapsing relation maps each node to that representative. The deleted edge set and deleted node set are empty.

Butterfly Edit. Compute the strongly connected components of the input graph. Choose a representative node from each component. The collapsing relation maps each node to the representative of the component to which it belongs. The deleted edge set and deleted node set are empty.

Reachable Edit. Given a reduced graph R, for each node $r \in R$ determine the contained node set of R relative to the original input graph G. Recall from Section 2 that we assume a heap snapshot associates an instance size attribute with each node. Compute the *uniquely-owned* size for each $r \in R$, which is the sum over each node $g \in G$ in the contained set of r of the instance size of g. Next, compute the *shared-owned* size for each node $r \in R$, which is the sum over all nodes r' reachable from r of uniquely-owned size of r'. Choose a threshold of this shared-owned property, a size below which would not be worth the effort of further study. We have found that a reasonable threshold is the maximum of one megabyte and one standard deviation above the mean size of all shared-owned sizes of the graph's nodes. The collapsing relation of this graph edit is the identity. The deleted edge set is empty. The deleted node set is those nodes whose shared-owned size falls below the threshold.

Miscellaneous Edit. Given a reduced graph R, determine the subset of the contained set of R relative to the original graph G that have been deleted; that is, those union of the contained set, relative to G, of nodes in a D_n of some reduced graph on the chain between G and R. We term this the "miscellaneous" set. Compute the sum M of the instance sizes of the members of the miscellaneous set. Compute the shared-owned size, S_r of each nodes $r \in R$. Choose a fraction ϵ of M so that the deleted node set of this graph edit is the set of nodes of $r \in R$ with $S_r - M < \epsilon$; this isolates any node that is responsible for only a



(a) jess (b) db (c) mtrt

(d) `A3`

Fig. 4. Our implementation automatically generates these ownership graphs

small amount of space, i.e. ϵM, on top of the miscellaneous size. The collapsing relation is the identity, and the deleted edge set is empty.

The Ownership Graph Algorithm. The ownership graph is the reduced graph resulting from the final edit in a chain of graph edits. We will need to apply certain edits more than once, because one edit may reintroduce a structure that another edit aggregates. Consider a variant of the graph of Figure 2(d), where one node in each of the two dominator tree references a third dominator tree. A dominator edit produces a graph of three nodes. A butterfly edit of that graph aggregates the two butterfly-connected nodes into one. The resulting two-node graph has a single edge between the former butterfly and that node representing the third dominator tree. Reapplying the dominator edit produces a single-node graph. The chain of graph edits we use to produce an ownership graph is: dominator, context-identical, butterfly, dominator, reachable-threshold, miscellaneous-threshold, context-identical, and finally dominator.

We have implemented this algorithm, and it consistently and quickly produces small ownership graphs. Recall that the two threshold edits may populate a pseudo-aggregate (`miscellaneous`) that represents the memory that falls below the chosen threshold. When rendering an ownership graph, we introduce a second pseudo-node (`everything`), to represent the entire snapshot; it refers to every

Table 7. The size and time to compute ownership graphs

application	nodes in full graph	nodes in ownership graph	seconds to construct
A1	29,181,452	14	148
A2	20,952,120	15	98
A3	17,871,063	11	82
A4	4,391,183	3	26
A2'	4,289,704	19	24
A5	4,099,743	13	27
A6	3,757,828	3	18
mtrt	509,170	2	8
db	342,725	1	7
javac	316,857	8	8
jess	83,815	2	7
jack	37,949	1	7
mpegaudio	9,997	1	5
compress	7,696	1	6

root in the graph. Table 7 shows the size and time to compute[4] ownership graphs. We do not count the two pseudo-nodes towards an ownership graph's node count. The computation time figures include the code to compute the graph halos, a DFS spanning tree, the dominator tree, all of the graph edits, and the time to render the graph to SVG (scalable vector graphics). For application A3, the full graph has nearly 18 million nodes; the ownership graph, computed in 82 seconds, consists of 11 nodes.

Figure 4 shows the output, generated automatically, from three of the benchmarks and application A3. Our rendering code draws a stack of nodes whenever an ownership graph node represents a context-identical aggregate. Each node shows the uniquely-owned bytes ("base") represented by that aggregate. Each non-leaf node also show shared-owned bytes ("cum"). Finally, we color the nodes based on the source package that primarily contributes to that aggregate's base size: dark gray for customer code, light gray for framework code (such as the application server, servlet processing, XML parsing code), black for standard Java library code, and white for everything else.

4 Summarizing Content Within Trees

This section shows how to summarize the nodes within a tree [12, 9, 20, 21, 25], using the concept of backbones. A backbone in a tree is a mechanism whereby collections of objects grow or shrink. The backbone of a linked list is the chain of "element" objects that store the inter-element linkage; in this case, the backbone structure is recursive. Section 4.1 introduces a categorization of the contents of a data structures based on how the objects contribute to backbones. This categorization alone provides powerful, but *flat* summaries of a tree's content.

[4] On a 1.8GHz Opteron, using Sun's Linux 1.5.0_06 64-bit JVM and the `-server` flag.

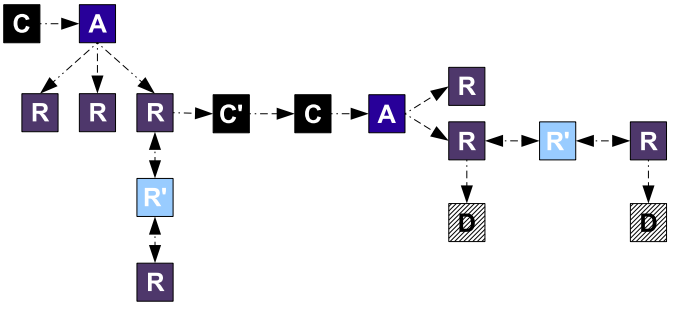


Fig. 5. A categorization of the nodes in a tree according to backbone structure

To summarize the *locations* of excessive memory footprint, Section 4.2 shows how to use a backbone categorization to aggregate backbones into equivalence classes, based on two notions of equality. We show that the equivalence relations successfully aggregate large number of backbones. Finally Section 4.3 shows how applying thresholds after having aggregated by backbone equivalence provides succinct summaries of tree content.

Note that, in some cases, a node in the ownership graph will be a dominator tree, and the approach described in this section applies directly. In other cases, it will be a collection of trees. To analyze a forest of trees, we take the union of the summaries of each tree.

4.1 The Elements of a Backbone

We identify six elements of a backbone within a tree, as shown in Figure 5. Array backbone types, those nodes labeled A, are responsible for horizontal growth or shrinkage in a graph. Recursive backbone types, nodes labeled R, can change the depth of a graph. We refer to the union of A and R types as *backbone types*. In some cases, a recursive backbone includes nodes of a non-backbone type (R′) that are sandwiched between the recursive backbone nodes. Above any backbone is a node that represents the container (C) to which they belong. There are often non-backbone nodes placed between one container and another, or between a backbone and a nested container; these *container sandwich* nodes are labeled C′. Underneath the backbone nodes, whether array or recursive, are the nodes that dominate the true data of the container (D). These six groups of types cover much of the structure within trees. We bundle any other structures not covered by the main six groups into the D group.

For example, an XML document can grow by adding elements or by adding attributes to an existing element. The elements grow recursively, but sometimes a `TextImpl` node is sandwiched between two `ElementImpl` nodes. The attributes grow along an array backbone, with data of type `AttributeImpl` under a container of type `Vector`. Between an element's recursive backbone and the `Vector` container is a container sandwich of type `AttributeMap`.

We categorize node types into one of these six groups. From this categorization of types, it is straightforward to categorize the nodes themselves. Array types

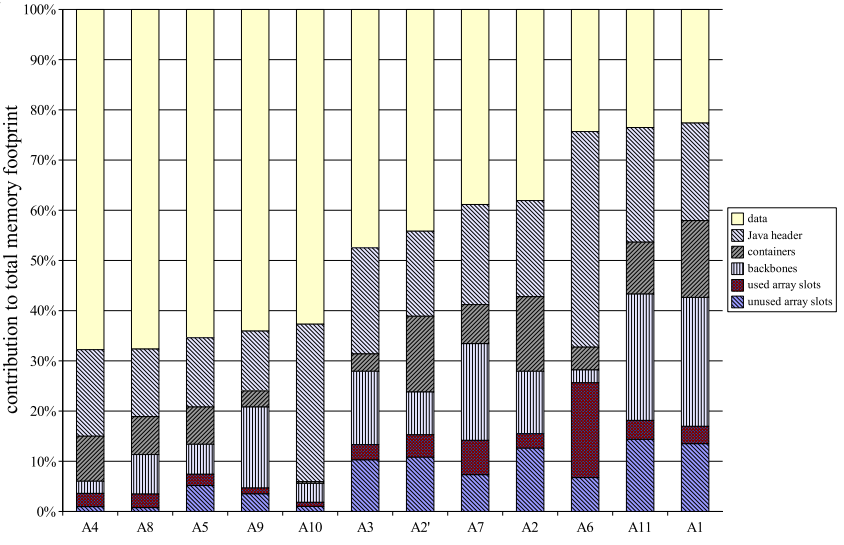


Fig. 6. The contribution of backbone overheads to total memory footprint

have instances that point to a number of nodes of the same type; the format of heap snapshots usually distinguishes array types for us. We currently identify only one-hop recursion, where nodes of a type point to nodes of the same type. This simple rule is very effective. Even in the XML document example of the previous paragraph, where there are recursive sandwich types, the recursive-typed nodes still point to nodes of the same type. A container type is a non-backbone type that has node instances that point to backbone types. Given a subpath in the tree that begins and ends with a node from R, all nodes between those endpoints are from R′. Given a subpath that begins with A, C, or R and that ends with C, all nodes between the endpoints are from C′. Finally, there will be a set of nodes that are pointed to by nodes of backbone type; the union of the types of nodes dominated by them form D.

Categorizing objects in this way yields powerful summaries of content, such as the ones shown in Figure 6. This figure includes five additional snapshots from real server applications, `A7-A11`, that we do not otherwise study in this paper. Each of the six categories in the figure represents the sum of the instance size of each node. We split the array backbone overhead into two subcategories: the memory devoted to unused slots in the array, and slots used for actual references. We assume that a `null` reference value in an array is not meaningful data; in our experience, this is nearly always the case. We also include the contribution of the Java object header, assuming eight bytes per header. We include header overhead, as its contribution varies for similar reasons as backbone overheads in general: many small collections leads to a higher C overhead, but also a higher object header overhead. We deduct this header cost from the costs associated with the other backbone overheads.

The amount of storage in the D group varies from as much as 68% to as little as 23%. On average, the data consumes 47% of the heap. This fraction is not well correlated with snapshot size; e.g. the snapshot A4 has over 20 million nodes, and yet has the highest fraction of data, while application A2', with a quarter the number of nodes, has a much lower fraction of data. Furthermore, the distribution to the various overheads is not constant: there is no hard and fast rule about how to impose a high backbone cost. It is certainly a property of the application; e.g. A2 and A2', which represent the same application in steady state, and just after server startup, have similar profiles. One application might have a few large data structures, versus many small ones; another might use an open-chained hashing implementation, rather than one with explicit chaining (the former would avoid a high R cost). Appendix A describes the data models we use in the implementations for this paper. Our layout nearly eliminates backbone and object header overheads, which is one of the ways we achieve high scalability.

4.2 Aggregates of Backbone Equivalence

Most real applications have a tremendous number of backbone nodes. As the second column of Table 8 shows, our real applications have anywhere from 67 thousand to 10 million distinct locations in their dominator trees that serve as either array or recursive backbones. This is far too many for a human to comprehend. Fortunately, there is much commonality in those backbone locations. We group the backbone nodes into equivalence classes, based on two equivalence properties: one based on type equality of paths and the second based on a notion of backbone-folding equality. While the second subsumes the first, to achieve a well-performing implementation, it is important to apply them one after the other, as computing context equality can be expensive.

Table 8. The number of backbone nodes and the number of root-type-path and backbone-folding equivalence classes (summed over all dominator trees)

application	backbone nodes	root-type-path equiv. classes	backbone-folding equiv. classes
A1	10,864,774	21,820	7,689
A2	4,704,630	23,561	10,381
A3	3,690,480	345,863	21,482
A2'	772,299	13,855	6,550
A4	342,570	9,630	5,046
A5	630,784	14,847	7,793
A6	107,802	3,907	1,840
mtrt	78,153	3,092	448
db	17,173	91	50
javac	116,274	18,818	9,025
jess	15,069	148	101
jack	2,690	289	154
mpegaudio	2,017	117	77
compress	1,985	175	48

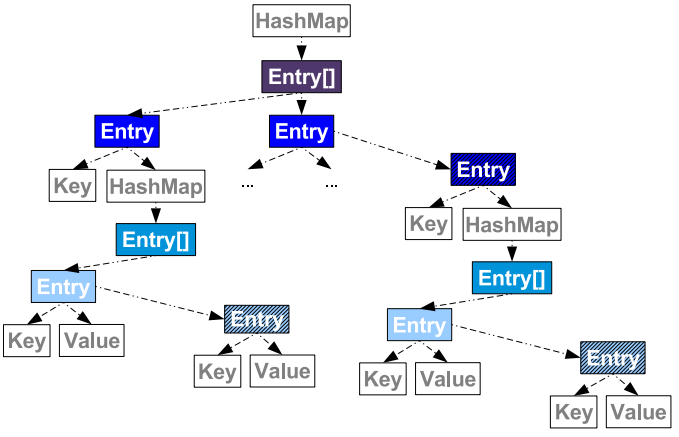


Fig. 7. A hash map of inner hash maps. There are two backbone types (`Entry[]` and `Entry`), ten nodes of those two types, nine backbone equivalence classes under root type path equality, and four backbone-folding equivalence classes.

Root-type-path Equality. Let the *root path* of a tree node be the list of nodes from the tree root to that node, inclusive; the *root type path* is similarly the list of those node types. We compute the A and R node types, and the instances of those types in the tree under analysis. We then form equivalence classes of these instances, using root type path equality.

It is often the case that a large number of backbone nodes in a tree have equal root type paths. Forming equivalence classes based on this notion of equality can therefore produce a more succinct summary of a tree's content than an enumeration of the backbone nodes. The third column in Table 8 shows the number of root type path equivalence classes in a number of applications and benchmarks.

Consider the common case of root type path equality shown in Figure 7: a hash map of inner hash maps, where all of maps use explicit chaining. There are two backbone types (`Entry[]` and `Entry`) and ten backbone nodes. Of those ten, there are nine distinct classes of backbone nodes under root type path equality. The only non-singleton class has the two top-left `Entry` nodes. Every other backbone node has a unique root type path. For example, the third `Entry` in the upper hash map is located under an `Entry` object, a property that the other two `Entry` nodes do not have. This difference skews every other node instance under that chained `Entry`, rendering little root type equivalence. We chose this example for illustrative purposes only. In practice, we see that from Table 8 that there are quite a large number of backbone nodes with type-identical root type paths. The figures in this table represent the sum over all dominator trees in each heap snapshot.

Backbone-folding Equality. Root type path equality identifies nine backbone equivalence classes in the tree of Figure 7. We feel there should only be four distinct classes. The upper `Entry[]` array is rightly in a singleton class, but the three upper `Entry` instances, the two lower `Entry[]` instances, and the four lower `Entry` instances should form a total of three classes, respectively. Imagine

that the lower hash maps contain values of type string: a hash map of hash maps that map to string values. We feel that each of those strings should be the same, despite being located in potentially thousands of separate (lower) hash maps, and despite each lower hash map being under a wide variety of depths of (upper) `Entry` recursion.

To capture this finer notion of equivalence, we observe that it is recursive structures, through combinations of R and R′ nodes, that lead to the kind of skew that foils root type path equality. We compute the set of A, R, and R′ nodes and, to each backbone node, associate a regular expression. The canonical set of regular expressions form the equivalence classes (c.f. the RDS types and instances of [32] and the DSGraphs of [20, 21]). The regular expression of a node is its root type path, except that any R′ node is optional and any R node can occur one or more times in any position. For example, the two D nodes from Figure 5 are backbone-folding equivalent, because they share the regular expression $CAR^+(R'?)CARL^+(R'?)$, where + and ? have the standard meanings of "one or more" and "optional".

The fourth column in Table 8 shows the number of equivalence classes of backbone nodes under backbone-folding equality. Even for graphs with tens of millions of nodes, aggregation alone (i.e. without filters or thresholds) collapses all dominator trees down to at most 21 thousand backbone equivalence classes.

4.3 Using Thresholds to Identify Large Patterns in Large Trees

Applying thresholds after having aggregated backbones can yield succinct summaries of content. As a first threshold, we usually care to study only the largest trees, or at least to study the largest trees first. Within a large tree, we consider two useful thresholds of backbone aggregates: a biggest contributing pattern analysis, and a suspect locator analysis.

A *biggest contributing pattern* analysis looks for the equivalence classes that contribute most to a given tree. Table 9 shows the result of a biggest-contributor analysis to the largest dominator tree in each application and benchmark. There are often hundreds of equivalence classes within the largest tree. However, only a few patterns summarize a substantial fraction of the footprint of the tree. The third column in the table shows how many of those equivalence classes account for 80% of the size of the tree (tabulating the largest equivalence classes first). With just two exceptions, a small handful of classes account for most of the footprint of the largest tree. Even for the two exceptions, `A1` and `javac`, 80% of the largest tree's size is accounted for by 35 and 53 patterns.

Sometimes, it is helpful to know where certain suspicious data types are placed in an expensive tree. A *suspect locator* analysis identifies the distinct classes of locations in which a chosen data type occurs. There may be millions of instances of this type, but they will not be in a million different equivalence classes. Furthermore, as Table 10 shows, for all of our applications and benchmarks, a only a handful of equivalence classes account for most of the contribution of that type in any one tree. This is despite the fact that, in some cases, there are hundreds of distinct patterns in which the largest data type is located. More generally,

Table 9. A biggest contributing pattern analysis shows that a few hot patterns account for 80% of the largest dominator tree's memory footprint

application	equiv. classes in largest tree	80% contributors in largest tree
A1	761	35
A2	20	10
A3	1	1
A2'	11	2
A4	2	1
A5	1172	3
A6	77	3
mtrt	43	10
db	1	1
javac	566	53
jess	1	1
jack	22	1
mpegaudio	2	1
compress	2	1

this suspect locator analysis can apply to other notions of suspects, such as the major contributors to backbone overhead: if my C overhead is so high, then tell me the patterns that contribute most. We will explore this more general form of analysis in future work.

5 Related Work

Techniques that *summarize* the internal structure of heap snapshots are relatively uncommon. Recent work [27, 29] introduces a system for counting, via general queries, both aggregate and reachability properties of an object reference graph. They have also done insightful characterization studies [28, 30]. Another recent work [25], akin to [12], summarizes reachability properties for each root in the graph. To our knowledge, these works do not aggregate the internal structural of the graphs according to context. Other related domains include:

Shape Analysis. Static shape analysis builds conservative models of the heap at every line of code [12, 9, 19, 20, 21]. They often use abstract interpretation to form type graphs (such as the RSRSG [9] or the DSGraph [20]); these summaries capture recursive structures, somewhat analogous to the regular expressions we form in Section 4.2. The work we discussed above [25] can be thought of as a kind of dynamic shape analysis.

Heap Profiling. This phrase usually applies to techniques that track the object allocations of an application for a period of time [6, 36, 34, 35, 37, 40, 32]. Mostly, the allocation site profiles are used to populate aggregate call graphs, and interpreted as one would a profile of execution time. Sometimes, the data is used to help design garbage collectors [15]. Some works combine static shape analysis with dynamic profile collection [32].

Table 10. A suspect locator analysis shows that a few hot patterns contain 80% of the bytes due to instances of the dominant data type

application	equiv. classes in largest tree	80% contributors classes in largest tree
A1	427	14
A2	7	1
A3	33	2
A2'	2	2
A4	1	1
A5	248	5
A6	6	3
mtrt	13	7
db	1	1
javac	1	1
jess	1	1
jack	1	1
mpegaudio	1	1
compress	1	1

Ownership Types. There is a large body of recent work on representing the ownership of objects in the static type system [8, 26, 4, 2, 7, 5, 1, 11, 24]. Some recent refinements have addressed issues such as sharing [26] and dominance [7]. The primary goal of this work is to enable better static analysis, such as less conservative alias analysis, or catching deadlocks at compile time [4].

Leak Analysis. An application that leaks memory will eventually be found to have an excessive memory footprint. Much of the prior work on memory leak detection either focuses on identifying allocation sites [13, 43, 42, 38, 3, 18], or on mostly-manual heap snapshot differencing [10, 31]. Our previous work [23] analyzes a pair of heap snapshots, and automates the detection of the heads of possibly leaking data structures. It neither address shared ownership, nor how to summarize the content underneath the leaking structures.

Visualization. The work of [14] introduces the idea of using the dominator tree to visualize object ownership. They also provide an clever composition of trees that mirrors the stack of activation records. In a similar vein, [33] presents an alternative visualization strategy that takes into account object references, domination, and characteristics of object usage. Similar to our previous work [23], they use heuristics to impose an ownership tree on a graph. None of these summarize nodes; by using the dominator spanning tree, they do filter out edges. Other tools require a human to browse what are essentially raw object reference graphs [10, 31, 3]. In some cases, these tools aggregate, but only locally; e.g. [10] aggregates outgoing edges by the referred-to type. Many tools also provide flat summaries that aggregate graph nodes by type, size, etc. The work of [25] includes a visualization component that describes reachability-from-root and age properties of objects in a heap snapshot, but concedes that it does not scale to graphs much larger than several thousand nodes.

6 Future Work

We see three exciting areas of future work. First, Section 4.3 demonstrated how to locate the patterns that explain the hottest elements of a flat summary by type. This is a powerful style of analysis, and we can extend it to be driven by a more general notion of suspects. For example, we can use it to locate the few patterns that explain most of the Java object header overhead. We can also introduce new kinds of suspects, such as large base class overhead.

Second, the ownership graph provides a visual representation of responsibility. We feel that there is a need for schematic visual representations of content. The backbone equivalence classes provide a good model for this summary. There is much work to be done in finding the powerful, yet concise, visual metaphors that will capture these patterns.

Third, we feel that the methodology employed in this paper, and the ownership structures we have identified can be useful in understanding the structure graphs from other domains. For example, many of the difficult aspects of graph size (scale, scattering of suspects in disparate locations in a graph, sharing of responsibility) have analogs in the performance realm. In performance, flat summaries usually only point out leaf methods, and yet the structure of a call graph is highly complex. We will explore this synergy.

7 Conclusion

It is common these days for large-scale object-oriented applications to be developed by integrating a number of existing frameworks. As beneficial as this may be to the development process, it has negative implications on understanding what happens at run time. These applications have very complicated policies governing responsibility and object lifetime. From a snapshot of the heap, we are left to reverse engineer those policies. On top of that, even uniquely-, non-contingently-owned objects have complex structure. Data structures that are essentially trees, like XML documents, are large, and represented with a multitude of non-tree edges. The common data types within them may be scattered in a million places; e.g. the attributes of an XML document's elements occur across the width and throughout the depth of the tree.

We have presented a methodology and algorithms for analyzing this web of complex ownership structures. In addition to their usefulness for summarizing memory footprint, we hope they are helpful as an exposition of the kinds of structures that occur in large-scale applications. Work that tackles object ownership from viewpoints other than runtime analysis may benefit from this study.

Acknowledgments

The author thanks Glenn Ammons, Herb Derby, Palani Kumanan, Derek Rayside, Edith Schonberg, and Gary Sevitsky for their assistance with this work.

References

1. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: The European Conference on Object-Oriented Programming. Volume 3086 of Lecture Notes in Computer Science., Oslo, Norway, Springer-Verlag (2004)
2. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: Object-oriented Programming, Systems, Languages, and Applications. (2002)
3. `Borland Software Corporation`: OptimizeIt™ Enterprise Suite. `http://www.borland.com/us/products/optimizeit` (2005)
4. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: Object-oriented Programming, Systems, Languages, and Applications. (2002)
5. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Symposium on Principles of Programming Languages. (2003)
6. Clack, C., Clayman, S., Parrott, D.: Lexical profiling: Theory and practice. Journal of Functional Programming **5**(2) (1995) 225–277
7. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: The European Conference on Object-Oriented Programming. Volume 2743 of Lecture Notes in Computer Science., Springer-Verlag (2003) 176–200
8. Clarke, D.G., Noble, J., Potter, J.M.: Simple ownership types for object containment. In: The European Conference on Object-Oriented Programming. Volume 2072 of Lecture Notes in Computer Science., Budapest, Hungary, Springer-Verlag (2001) 53–76
9. Corbera, F., Asenjo, R., Zapata, E.L.: A framework to capture dynamic data structures in pointer-based codes. IEEE Transactions on Parallel and Distributed Systems **15**(2) (2004) 151–166
10. De Pauw, W., Sevitsky, G.: Visualizing reference patterns for solving memory leaks in Java. Concurrency: Practice and Experience **12** (2000) 1431–1454
11. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Special Issue: ECOOP 2004 Workshop FTfJP, Journal of Object Technology **4**(8) (2005) 5–32
12. Ghiya, R., Hendren, L.J.: Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In: Symposium on Principles of Programming Languages. (1996)
13. Hastings, R., Joynce, B.: Purify — fast detection of memory leaks and access errors. In: USENIX Proceedings. (1992) 125–136
14. Hill, T., Noble, J., Potter, J.: Scalable visualizations of object-oriented systems with ownership trees. Journal of Visual Languages and Computing **13** (2002) 319–339
15. Hirzel, M., Hinkel, J., Diwan, A., Hind, M.: Understanding the connectivity of heap objects. In: International Symposium on Memory Management. (2002)
16. Hitchens, R.: Java NIO. First edn. O'Reilly Media, Inc. (2002)
17. Holzner, S.: Eclipse. First edn. O'Reilly Media, Inc. (2004)
18. `IBM Corporation`: Rational PurifyPlus (2005)
19. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. In: International Static Analysis Symposium. Lecture Notes in Computer Science, New York, NY, Springer-Verlag (2004)
20. Lattner, C., Adve, V.: Data structure analysis: A fast and scalable context-sensitive heap analysis. Technical Report UIUCDCS-R-2003-2340, Computer Science Deptartment, University of Illinois (2003)

21. Lattner, C., Adve, V.: Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In: Programming Language Design and Implementation, Chicago, IL (2005) 129–142
22. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flow graph. ACM Transactions on Programming Languages and Systems **1**(1) (1979) 121–141
23. Mitchell, N., Sevitsky, G.: Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In: The European Conference on Object-Oriented Programming. Volume 2743 of Lecture Notes in Computer Science., Springer-Verlag (2003) 351–377
24. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Symposium on Principles of Programming Languages. (2005)
25. Pheng, S., Verbrugge, C.: Dynamic shape and data structure analysis in java. Technical Report 2005-3, School of Computer Science, McGill University (2005)
26. Pollet, I., Charlier, B.L., Cortesi, A.: Distinctness and sharing domains for static analysis of Java programs. In: The European Conference on Object-Oriented Programming. Volume 2072 of Lecture Notes in Computer Science., Springer-Verlag (2001) 77–98
27. Potanin, A.: The Fox — a tool for object graph analysis. Undergraduate Honors Thesis (2002)
28. Potanin, A., Noble, J., Biddle, R.: Checking ownership and confinement. Concurrency and Computation: Practice and Experience **16**(7) (2004) 671–687
29. Potanin, A., Noble, J., Biddle, R.: Snapshot query-based debugging. In: Australian Software Engineering Conference, Melbourne, Australia (2004)
30. Potanin, A., Noble, J., Frean, M., Biddle, R.: Scale-free geometry in object-oriented programs. In: Communications of the ACM. (2005)
31. `Quest Software`: JProbe® Memory Debugger. `http://www.quest.com/jprobe` (2005)
32. Raman, E., August, D.I.: Recursive data structure profiling. In: ACM SIGPLAN Workshop on Memory Systems Performance. (2005)
33. Rayside, D., Mendel, L., Jackson, D.: A dynamic analysis for revealing object ownership and sharing. In: Workshop on Dynamic Analysis. (2006)
34. Rojemo, N., Runciman, C.: Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In: International Conference on Functional Programming. (1996) 34–41
35. Runciman, C., Rojemo, N.: New dimensions in heap profiling. Journal of Functional Programming **6**(4) (1996) 587–620
36. Sansom, P.M., Peyton Jones, S.L.: Time and space profiling for non-strict higher-order functional languages. In: Symposium on Principles of Programming Languages, San Francisco, CA (1995) 355–366
37. Sansom, P.M., Peyton Jones, S.L.: Formally based profiling for higher-order functional languages. ACM Transactions on Programming Languages and Systems **19**(2) (1997) 334–385
38. Shaham, R., Kolodner, E.K., Sagiv, M.: Automatic removal of array memory leaks in java. In: Computational Complexity. (2000) 50–66
39. Shaham, R., Kolodner, E.K., Sagiv, M.: Estimating the impact of heap liveness information on space consumption in Java. In: International Symposium on Memory Management. (2002)
40. Shaham, R., Kolodner, E.K., Sagiv, S.: Heap profiling for space-efficient java. In: Programming Language Design and Implementation. (2001) 104–113
41. `SPEC Corporation`: The SPEC JVM Client98 benchmark suite. `http://www.spec.org/osg/jvm98` (1998)

42. Sun Microsystems: Heap Analysis Tool. https://hat.dev.java.net/ (2002)
43. Sun Microsystems: HPROF JVM profiler. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html (2005)

A Modeling Gigantic Graphs

To program graph analysis algorithms in Java, we must be careful to avoid our own memory footprint problems. We could easily find ourselves modeling the Java heap of a large server inside the Java heap on a development machine. To write scalable graph analysis algorithms in Java, we made two implementation decisions. We do not store graphs in an object-oriented style. Instead, we represent node attributes and edges as columns of data, and store each column as a separate file on disk. There is no `Node` data type. Rather, code refers to nodes as 32-bit integer identifiers, ranging densely from 0 to the number of nodes; the same is true for the edges (limiting us to two billion nodes). This storage layout avoids an object header for each node, and avoids any container cost to represent the outgoing and incoming edges for each node.

In addition to lowering footprint requirements, this style of storage aids performance. It permits direct use of the `java.nio` package [16] to memory map attributes on demand. This gives us constant time reloading of graphs, transparent persistence of graphs and attributes, the operating system takes care of caching for us (even across process boundaries), and we can run any analysis with the default Java heap size, independent of the size of the graph under analysis.[5]

[5] `java.nio` is not without its faults; e.g. it currently lacks an explicit unmap facility.