Patterns of Memory Inefficiency

Adriana E. Chis[1], Nick Mitchell[2], Edith Schonberg[2], Gary Sevitsky[2], Patrick O'Sullivan[3], Trevor Parsons[1], and John Murphy[1]

[1] University College Dublin, Dublin, Ireland
{adriana.chis,trevor.parsons,j.murphy}@ucd.ie
[2] IBM T.J. Watson Research Center, Hawthorne, NY, US
{nickm,ediths,sevitsky}@us.ibm.com
[3] IBM Software Group, Dublin, Ireland
patosullivan@ie.ibm.com

Abstract. Large applications often suffer from excessive memory consumption. The nature of these heaps, their scale and complex interconnections, makes it difficult to find the low hanging fruit. Techniques relying on dominance or allocation tracking fail to account for sharing, and overwhelm users with small details. More fundamentally, a programmer still needs to know whether high levels of consumption are *too* high.

We present a solution that discovers a small set of high-impact memory problems, by detecting patterns within a heap. Patterns are expressed over a novel *ContainerOrContained* relation, which overcomes challenges of reuse, delegation, sharing; it induces equivalence classes of objects, based on how they participate in a hierarchy of data structures. We present results on 34 applications, and case studies for nine of these. We demonstrate that eleven patterns cover most memory problems, and that users need inspect only a small number of pattern occurrences to reap large benefits.

Keywords: memory footprint, memory bloat, pattern detection, tools.

1 Introduction

In Java, applications can easily consume excessive amounts of memory [13]. We commonly see deployed server applications consume many gigabytes of Java heap to support only a few thousand users. Increasingly, as hardware budgets tighten, memory per core decreases, it becomes necessary to judge the appropriateness of this level of memory consumption. This is an unfortunate burden on most developers and testers, to whom memory consumption is a big black box.

We have spent the past two years working with system test teams that support a family of large Java applications. These teams perform extensive tests of applications, driving high amounts of load against them. While running these tests, they look at gross measures, such as maximum memory footprint. They may have a gut feeling that the number is high, but have little intuition about whether easy steps will have any measurable impact on memory consumption. Sizing numbers alone, whether memory consumption of the process, of the heap,

M. Mezini (Ed.): ECOOP 2011, LNCS 6813, pp. 383–407, 2011.

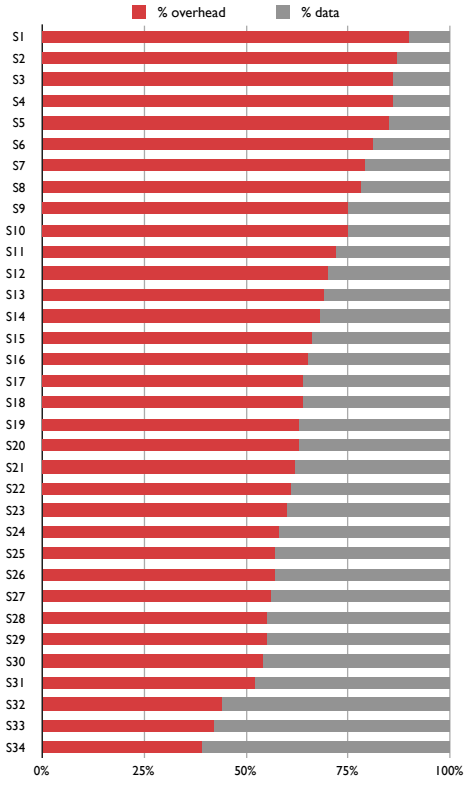


Fig. 1. The fraction of the heap that is overhead, including JVM object headers, pointers used to implement delegation, and null pointer slots, is often surprisingly high

and even of the size of individual data structures [3,18,10], are not sufficient. The test teams need a quick evaluation of whether deeper code inspections will be a worthwhile investment of time.

If size alone does not indicate appropriateness or ease of remediation, then perhaps measures of overhead can. Prior work infers an overhead measure, by distinguishing the *actual data* of a data structure from the implementation costs necessary for storing it in a Java heap [13]. Overheads come from Java Virtual Machine (JVM) object headers, null pointers, and various overheads associated with collections, such as the `$Entry` objects in a linked structure. Fig. 1 shows this breakdown, of actual data versus overhead, for 34 heap snapshots from 34 real, deployed applications. The figure is typically quite high, with most snapshots devoting 50% or more of their heap to implementation overheads.

Unfortunately, when presented with this figure, even on a per-structure basis, these testing teams were left with only a modified form of their original dilemma. Instead of wondering how large is too large, they now asked how much overhead is too much. To a development team, if a memory footprint problem is complicated to fix, or if the fix is difficult to maintain, it is of little value. Depending on the

nature of the overhead, a data structure may be easy to tune, or it may just be something the team has to live with. There are, after all, deadlines to be met.

An Approach Based on Memory Patterns. We have found that common design mistakes are made, across a wide range of Java applications. For example, it is common for developers to use the default constructors when allocating collections: `new ArrayList()`. If the code only stores a few items in these lists, we consider this to be an occurrence of a *sparse collection* pattern; only a few of the allocated pointers are used, thus the overhead comes from the empty slots. Hashmaps are often nested, and, if the inner maps are very small, this is an occurrence of the *nested small collection* pattern. These sound straightforward, and that was our goal: find problems that are easily understood, and easily fixed. Even if we miss large swaths of overhead, at least we are catching the easy stuff.

We discovered that this approach, when based on the right set of patterns, reliably explains a majority of overheads across a wide diversity of applications.

Detecting and Summarizing Pattern Occurrences. The challenges of this work came in two parts. First was the cataloging process. This involved a year of combing over data from many hundreds of real applications, to find the important patterns. The second challenge lay in the detection and reporting of pattern occurrences in complex heaps. Existing abstractions and analyses are insufficient to detect many of the common patterns. Knowing that items stored in a collection suffer from a high amount of delegation (a design that is actively encouraged [8]), with its attendant header and pointer overheads, requires knowing the *boundaries* of those items. Where would a scan of the graph of objects start, and where would it stop, in order to know that these items are highly delegated, and hence report the cost of this occurrence?

The choice of aggregation is crucial for detecting and reporting problems. Our approach is motivated by two properties prevalent in large-scale Java programs. First, multiple instances participate in larger cohesive units, due to the high degree of delegation common in the implementations of containers and user-defined entities. All of the objects in such a highly delegated design are grouped under a single occurrence of a larger pattern. We show how an aggregation by the *role* objects play in a data structure can ensure that we detect these larger patterns. Objects are either part of collection infrastructure, or part of the implementation details of contained items (entities or nested collections).

Furthermore, framework- and user-defined structures are frequently reused in multiple contexts. Frameworks themselves employ other frameworks, leading to deeply layered designs. Existing approaches aggregate by allocation context [7,16], or by only one hop of points-to context [2,18,3]. To generate meaningful reports, aggregation by deep context is important, in order to distinguish different uses (or misuses) of the same building blocks. The analysis cannot report millions of similar pattern occurrences, for example, one for each of the inner maps in a nest of hashmaps. In these cases, the context necessary to properly cluster can be in the dozens of levels of pointers.

The contributions of this paper are the following:

- Eleven commonly occurring patterns of memory misuse. Table 1 presents a histogram of the percentage of overhead explained by the eleven patterns, over 34 heap snapshots.
- The *ContainerOrContained Model* , a single abstraction that can be used both to detect occurrences of these patterns, and aggregate these occurrences into concise summaries based on the data structure context in which they occur. The abstraction defines the roles that data types play in the collection and non-collection implementation details of the application's data models.
- An analysis framework for detecting and aggregating *pattern occurrences*, and encodings of the patterns as client analyses.
- A tool that implements this framework, and evaluations of its output on nine heaps. This tool is in initial use by system test teams within IBM.
- A characterization study of footprint problems in 34 heaps. The study shows, for example, that our set of patterns suffice to explain much of the overhead in heaps, and that a tool user typically need only inspect a small number of pattern occurrences to reap large benefits.

Fig. 2 summarizes our approach. We acquire a heap snapshot from a running Java application. From this snapshot, we compute the ContainerOrContained Model. We have encoded the patterns as clients of a common graph traversal algorithm. The traversal of a chosen data structure computes the count and overhead of each pattern occurrence, aggregated by its context within the structure.

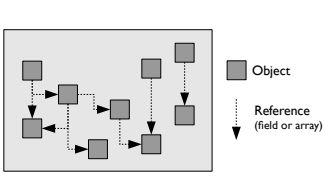
2 The Memory Patterns

We have found that eleven patterns of memory inefficiency explain the majority of overhead in Java applications. The patterns can be divided into two main groups: problems with collections, and problems with the data model of contained items. The goal of this section is to introduce the patterns. In the next sections, we introduce a set of data structure abstractions and an algorithm for detecting and aggregating the *occurrences* of these patterns in a given data structure.

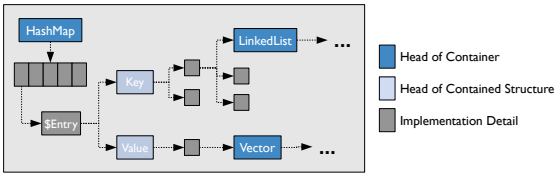
All of these patterns are common, and lead to high amounts of overhead. Table 2 names the eleven patterns. We use a short identifier for each, e.g. P1

Table 1. The analysis presented in this paper discovers easy to fix problems that quite often result in big gains. These numbers cover the heap snapshots in Fig. 1. The overhead is computed as described in Sect. 3.4.

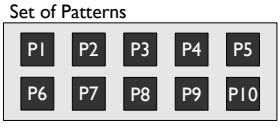
overhead explained	# applications
0–30%	0
30–40%	4
40–60%	9
60–80%	7
80–100%	14



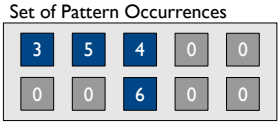
(a) A heap snapshot



(b) A ContainerOrContained Model instance for one data structure



(c) We present eleven memory pattern client analyses.



(d) The analyses scan for *occurrences* of the patterns; e.g. P1 occurred 3 times.

Fig. 2. From the raw concrete input, a heap snapshot from a Java application, we compute a set of abstract representations. We compute one abstract form, called the ContainerOrContained Model, per data structure in the heap snapshot. The client analyses scan each data structure for problematic memory patterns, making use this abstract form.

stands for the pattern of empty collections. Table 3 shows that these patterns do indeed occur frequently across our sample heap snapshots, often multiple times per snapshot. Sect. 5 gives detailed findings of our detection algorithm.

2.1 Patterns P1–P3: Empty, Fixed, Small Collections

Each of these patterns has the general nature of a large number of collections with only a few entries. This situation leads to a high amount of overhead due to a lack of amortization of the fixed costs of a collection. The fixed costs of a `HashSet` in the Java standard library, which includes multiple Java objects and many field slots, is around 100 bytes (on a 32-bit JVM). This sounds like an inconsequential number, but if that set contains only a few entries, then the relative contribution of that fixed overhead to the total heap consumption is high. The fixed cost of a `ConcurrentHashMap` in Java is 1600 bytes!

Two important special cases have very different remedies from the general case of small collections. The first is the fixed-size small collections pattern, where all the instances of such collections contain always the same constant number of entries. These may benefit from using array structures, rather than a general purpose collection. The second is the empty collections pattern; occurrences of these could be lazily allocated.

2.2 Pattern P4: Sparsely Populated Collections

Collections that have an array-based implementation risk being sparsely populated. Fig. 3 shows an `ArrayList` that was instantiated with its default size,

Table 2. The eleven memory patterns

memory pattern	identifier
Empty collections	P1
Fixed-size collections	P2
Small collections	P3
Sparsely populated collections	P4
Small primitive arrays	P5
Boxed scalar collections	P6
Wrapped collections	P7
Highly delegated structures	P8
Nested Collections	P9
Sparse references	P10
Primitive array wrappers	P11

typically 10 or 12 entries, but that currently contains only two Strings. Unlike the first three patterns, this pattern affects both a large number of small (sparse) collections, and a single large (sparse) collection. The causes of a poorly populated collections are either: 1) the initial capacity of the collection is too high, or 2) the collection is not trimmed-to-fit following the removal of many items, or 3) the growth policy is too aggressive.

2.3 Pattern P5: Small Primitive Arrays

It is common for data structures to have many small primitive arrays dangling at the leaves of the structure. Most commonly, these primitive arrays contain string data. Rather than storing all the characters once, in a single large array, the application stores each separate string in a separate `String` object, each of which has its own small primitive character array. The result is often that the overhead due to the header of the primitive character array (12 bytes, plus 4 bytes to store the array size) often dwarfs the overall cost of the data structure. If this data is intended to be long-lived, then it is relatively easy to fix this problem. Java `String`s already support this substring optimization.

Table 3. Across the 35 snapshots in Fig. 1, the memory patterns occur frequently. The patterns are also not spurious problems that show up in only one or two applications; many occur commonly, across applications.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
# pattern occurrences	37	16	45	11	7	19	2	111	5	5	46
# applications	18	12	20	8	6	13	2	29	3	4	19

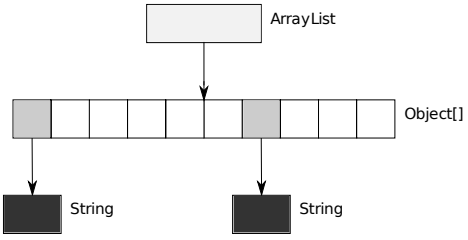


Fig. 3. An example of the sparse collection pattern, in this case with eight null slots

2.4 Pattern P6: Boxed Scalar Collections

The Java standard collections, unlike C++ for example, do not support collections with primitive keys, values, or entries. As a result, primitive data must be boxed up into wrapper objects that cost more than the data being wrapped. This generally results in a huge overhead for storing such data.

2.5 Pattern P7: Wrapped Collections

The Java standard library requires the use of wrappers to modify the behavior of a collections. This includes, for example, making a collection synchronized or unmodifiable. `HashSet` is implemented in this way, too: as a wrapper around `HashMap`. This is another case of a cost that would be amortized, if the collections had many entries, but one with a distinct remedy.

2.6 Pattern P8: Highly Delegated Structures

Java data models often require high degrees of delegation. For example, an employee has attributes, such as a name and email address. In Java, due to its single-inheritance nature, one is often forced to delegate the attributes to side objects; for example, the developer may wish to have these two attributes extend a common `ContactInformation` base class. The frequent result is a highly delegated web of objects, and the repeated payment of the object "tax": the header, alignment, and pointer costs.

2.7 Pattern P9: Nested Collections

This pattern covers the common case of nested collections. One can use a `HashMap` of `HashSet`s to model a map with multiple values per key. Similarly, a `HashMap` of `ArrayList`s can be used to represent a map which requires multiple objects to implement a key. Fig. 4 portrays a `HashMap` of `HashSet` where `String` key maps to a set of values, implemented using a `HashSet`. For this current paper, we only cover these two important cases of nested collections: `HashMap`s with either `HashSet` or `ArrayList` keys or values.

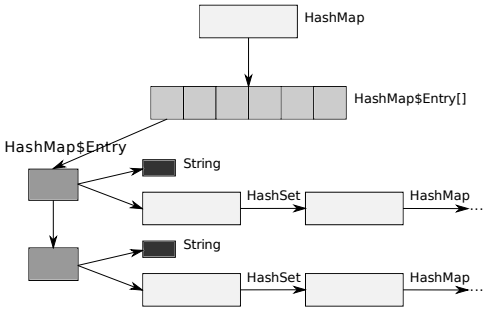


Fig. 4. An example of the HashMap of HashSet collection pattern

2.8 Pattern P10: Sparse References

In addition to high degrees of delegation, non-collection data often suffers from many null pointers. This pattern is an important special case of pattern P8: highly delegated structures. A highly delegated design can suffer from *overgenerality*. The data model supports a degree of flexibility, in its ability to contain extra data in side objects, that is not used in practice.

2.9 Pattern P11: Primitive Array Wrappers

The last non-collection overhead pattern comes from wrappers around primitive arrays. These include the `String` and `ByteArrayOutputStream` objects whose main goal is to serve as containers for primitive data. This is a cost related to P5: small primitive arrays, but one that is outside of developer control; hence we treat it separately. The Java language does not allow primitive arrays to be stored inline with scalar primitive data.[1] We include this pattern for completeness, even though in practice developers would have trouble implementing an easy fix to its occurrences. We wanted to include it, so that the characterization of Sect. 6 can motivate language and compiler developers to fix this problem.

3 The ContainerOrContained Abstraction

We introduce a single abstraction that is useful for both detecting occurrences of memory patterns and aggregating those occurrences in a way that concisely summarizes the problems in a data structure. We begin by describing the raw input to the system of this paper, and briefly present the important limitations of the dominator relation for heap analysis.

[1] Java supports neither structural composition nor value types, those features of C and C# that permit a developer to express that one object is wholly contained within another. At least it can be done manually, in the case of scalar data. This is simply not possible for array data.

3.1 Heap Snapshots and the Object Reference Graph

The system described in this paper operates on snapshots of the Java heap. This data is readily available from most commercial JVMs. Typically, one sends a signal to a Java process, at which point the JVM produces a file that contains the contents of the Java heap. The analysis can be done disconnected from any test runs, and the generated snapshots can be shared with development teams at a later date.

A heap snapshot can be considered as a graph of objects and arrays, interconnected in the way that they reference each other. Fig. 2(a) shows a small-scale picture of how we consider the heap to be a graph. This graph is commonly referred to as an *object reference graph*.

3.2 Limitations of the Dominator Relation

Several existing tools [3,18] base their visualizations and analyses on the *dominator forest* [9], rather than the full graph. This was also our first choice; it has some enticing qualities. When applied to an object reference graph, the dominator relation indicates unique ownership: the only way to reach the dominated object is via reference chasing from the dominator.

Unfortunately, the dominator forest is a poor abstraction for memory footprint analysis, due to the issue of shared ownership. Fig. 5 illustrates a case where two data structures share ownership of a sub-structure. An analysis that requires counting the number of items in a collection, such as the linked-list style structure in Data Structure 1, must count all items, whether or not they are simultaneously part of other structures. A traversal of the dominator tree of Data Structure 1 will only count two items — the edge leading to the third is not a part of the dominator forest. In addition to failing to account for paths from multiple roots, the dominator relation also improperly accounts for diamond structures.

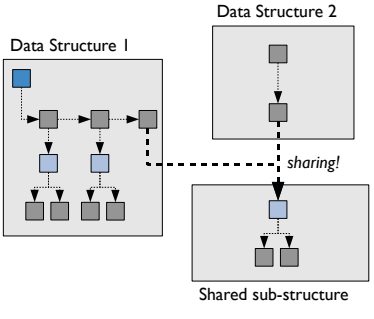


Fig. 5. The dominator forest is unhelpful both for detecting and for sizing memory problems. No traversal of a dominator tree (e.g. Data Structure 1 or 2) will encounter shared sub-structures. A collection, even if it dominates none of its constituents, should still be considered non-empty.

The dominator relation introduces a fake edge, from the root of the diamond to the tail. For these reasons, we base our analysis upon traversals of the full graph.[2]

3.3 Roles, and the ContainerOrContained Model

The bulk of objects in a data structure takes on one of six roles. These roles are summarized in Table 4. Consider a common "chained" implementation of a hashmap, one that uses linked list structures to handle collisions. Instances of this hashmap, in Java, are stored in more than one object in the runtime heap. One of these will be the entry point to the collection, e.g. of type `HashMap`, and the rest of the objects will implement the details of the linking structure. These two roles, the *Head of Container*, and *Implementation Details*, are common to most implementations of structures that are intended to contain an indefinite number of items.

Underneath the chains of this hashmap will be the contained data structures. These constituent structures have a similar dichotomy of roles: there are the *Head of Contained* structures, and, for each, the implementation details of that contained structure. Consider the example from earlier (Sect. 2.6): an `Employee` data structure that has been implemented to delegate some of its functionality to other data types, such as `PhoneNumber` and `EmailAddress`. That these latter two pieces of information have been encoded as data types and hence (in Java) manifested as objects at runtime, is an implementation detail of the `Employee` data structure. Another role comes at the interface between the container's implementation details and the head of the contained items. For example, in a chained hashmap, the "entry" objects (`HashMap$Entry` in the Java standard collections library) will serve the role as this *Container-Contained Transition* objects. This role is crucial to correctly detect some of the patterns (shown in Sect. 4). The final important role, *Points to Primitive Array*, corresponds to those objects that serve as wrappers around primitive arrays.

We introduce the ContainerOrContained abstraction, that assigns each object in a data structure to at least one of these six roles. Objects not stored in a collection are unlikely to be the source of memory problems, and hence do not receive a role in this model. Given an object that is at the root of a data structure, we show how to compute that structure's ContainerOrContained model.[3] First, data types that form linking structures, such as the "entry" objects in a chained hashmap, are identified by looking for cycles in a points-to graph over types (a simple technique first described in [10]). Any instances of these types are assigned the Transitionary role, and objects they reference are assigned the Head of Contained role. Any arrays of references that don't point to a Transitionary object are themselves Transitionary; any objects these

[2] Many tools, including the authors' previous work, made a switch over to using the dominator relation. The demos for one [3] even claim it as their "secret sauce".

[3] In the case that there is a connected component at the root of the data structure, choose any object from that cycle.

arrays point to are Heads of Contained structures. Finally, objects that point to arrays of references or recursive types are Heads of Containers. Note that it is possible for an object to serve multiple roles, simultaneously. A `HashMap` inside a `HashMap` is both Head of Container and Head of Contained. A `String` key is both Head of Contained, and it Points to a Primitive Array. The remaining objects are either the Implementation Details of a collection, or of the contained items.

Table 4. In the ContainerOrContained abstraction, objects serve these roles

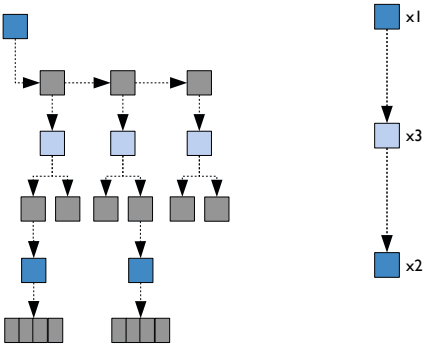
Role	Examples
Head Of Container	`HashMap`, `Vector`
Head Of Contained	keys and values of maps
Container-Contained Transition	`HashMap$Entry`
Points to Primitive Array	`String`
Collection Impl. Details	HashMap$Entry[]
Contained Impl. Details	everything else

3.4 How Roles Imply Per-Object and Total Overhead

The ContainerOrContained model defines a role for each object in a data structure. Given this mapping, from object to role, we show that one can compute the *total overhead* in that data structure; previous work [13] introduced this concept, and here we show a novel, much simpler way, to approximate total overhead using only a ContainerOrContained model that doesn't rely on dominance. The goal of this paper is to explain as much of that total overhead as possible, with a small number of pattern occurrences.

Definition 1 (Per-object Overhead, Total Overhead). *Let G be an object reference graph and $D \subseteq G$ be a data structure of G. The* total overhead *of D is the sum of the* per-object overhead *of each object in D. The per-object overhead of an object depends on its role:*

- **Entirely overhead***: if its role is Head of Container, Transitionary, or Collection Implementation Detail, then the portion is 100%.*
- **Headers and pointers***: if its role is Head of Contained or Contained Implementation Detail, then the portion includes only the JVM headers, alignment, and pointer costs.*
- **Headers, pointers, and primitive fields***: if its role is Points to Primitive Array, we also include primitive fields, under the assumption that many primitive array wrappers need to store bookkeeping fields, such as offsets, lengths, capacities, and cached hashcodes, that are not actual data.*



(a) A data structure (b) ContainerOrContained DAG

Fig. 6. A data structure, and its corresponding ContainerOrContained DAG

3.5 Regions, and the ContainerOrContained DAG

For every data structure, and the ContainerOrContained model over its constituent types, there is a corresponding directed acyclic graph (DAG) that summarizes the structure's contents [13]. Informally, the ContainerOrContained DAG is one that collapses nodes in an object reference graph according to the role they play and the context in which they are situated. Fig. 6 shows an example data structure and the corresponding ContainerOrContained DAG. Observe how this structure has a two-level nesting of collections: the outer collection has a linking structure, and the inner map has an array structure. Sandwiched between the two collections is a contained item that delegates to two sub-objects; one of the sub-objects contains the inner array-based collection. The ContainerOrContained DAG collapses this structure down to a, in this case, tree with three nodes. In general, this summarized form will be a DAG, in the case of diamond structures.

We define the ContainerOrContained DAG according to an equivalence relation of object reference graph nodes. We present a novel definition and construction algorithm that shows how this DAG is directly inferrable from the ContainerOrContained model, without reliance on dominance. First, the nodes of a DAG are *regions*, which represent one of the two primary roles:

Definition 2 (Head of Region). *Let G be an object reference graph and C be a ContainerOrContained model of the types of G. We say that a node $n \in G$ is a* Head of Region *if the role of n under C is either Head of Container or the Head of Contained.*

From this, the equivalence relation is defined as follows:

Definition 3 (ContainerOrContained Equivalence). *Let G be an object reference graph, $D \subseteq G$ be a data structure in G with back edges pruned, and C*

be the ContainerOrContained model for the types of G. Two nodes $n_1, n_2 \in D$ are equivalent under C if either 1) n_1 is a head of region and n_2 is not and there is no intervening head of region n_3 between n_1 and n_2 (or vice versa for n_1 and n_2); or 2) neither n_1 nor n_2 is a head of region, but they lie under an n_3 that is a head of region, with no intervening head of region n_4 that is on either the path from n_1 to n_3 or from n_2 to n_3; or 3) n_1 and n_2 have the same role under C and the parents of n_1 are equivalent to the parents of n_2.

4 Detecting Pattern Occurrences, and Aggregating Them by Context

We describe an algorithm that, parameterized by the details of a pattern, scans for occurrences of that pattern in a given data structure.[4] We initially coded each pattern as its own set of code, but eventually came to realize that each pattern differed in only three ways:

- The `start` and `stop` criteria. The boundaries of an occurrence, the details of which vary from pattern to pattern, but can always be expressed in terms of roles. For example, whether a `HashMap` instance is an occurrence of the empty collection pattern depends on the objects seen in a traversal of the subgraph bounded on top (as one traverses from the roots of the data structure being scanned) by a Head of Container, and bounded on the bottom by the Heads of Contained items.
- The `accounting` metrics. Each pattern differs in what it needs to count, as the scan proceeds. The empty collection pattern counts the number of Heads of Contained. The sparse references pattern counts a pair of numbers: the number of valid references, and the number of null slots.
- The `match` criterion. The empty collections pattern matches that `HashMap` if the number of Heads of Contained objects encountered is zero.

Observe that the empty collections pattern cannot count the number of Transitionary objects, (`HashMap$Entry` in this case), for two important reasons: 1) because some collections use these Transitionary objects as sentinels; and 2) sharing may result in two Transitionary objects referencing a single Head of Contained object.

Each match of a pattern would, without any aggregation, result in one pattern occurrence. This would result in needlessly complicated voluminous reports. Instead, as the algorithm traverses the data structure, it detects which ContainerOrContained region it is current in. Any occurrences of a pattern while the traversal is in a particular region are aggregated into that region. The output is a set of encountered regions, each with a set of occurrences. Each occurrence will be sized according to the accounting metrics of the pattern. Fig. 7 gives Java-like pseudocode for the algorithm.

[4] A set of patterns can be scanned for simultaneously. The description in this section can be generalized straightforwardly to handle this.

For example, a scan for occurrences of the empty collections pattern would count the total overhead of the collection's Implementation Details, matches if the number of Heads of Contained is zero, and upon a match accumulate that overhead into the ContainerOrContained region in which the empty collection is situated.

4.1 Client Analyses

Each pattern is detected by a client analysis of the general algorithm of Fig. 7. In this paper, we chose three patterns that were illustrative of the interesting variations from client to client. Sect. 5 then describes an implementation, in which all eleven clients have been implemented.

P3: Small Collections. This client is activated at a Head of Container and deactivated at a Head of Contained. The client counts the per-object overhead of the collection's Implementation Details and the number of Head of Contained encountered. The accounting operation plays an important role in the pattern detection. In the case of a map, unless done careful, the client would double count the key and the value (recall that the Transitionary element points to a key and a value). We do so by deactivating the accounting when a Head Of Contained is reached. The scanning is reactivated at the traversal in postorder of the Transitionary object. A collection instance, e.g. one instance of a `HashMap`, matches the pattern if it contains, in the current implementation, at most nine entries.

P4: Sparsely Populated Collections. The scope of a sparsely populated collection is from a Head of Container element up to the next Head of Container or Head of Contained element. The pattern counts the number of null slots and its corresponding overhead and the number of non-null slots. The pattern matches the collection when the number of null slots is greater than the number of non-null slots.

P5: Small Primitive Arrays. The boundary of a small primitive array pattern occurrence is a Points to Primitive Array elements as start condition and the traversal stops when the primitive array is reached. The client counts the per-object overhead of the primitive array and the size of actual primitive data. A match happens when the amount of data is small, compared to the overhead.

Table 5 presents the time to detect[5] all the eleven pattern categories in the heap snapshots from Fig. 1. While computation time is sometimes less than 2 minutes for heaps with tens of million of objects (e.g. Applications S8 and S9, with 57 and 34 million of objects respectively), there are also extreme cases where the computation time is high. Currently we are working on a few optimization possibilities to address the slow analysis time.

[5] Using Sun's Linux JVM 1.6.0_13 64-Bit with -server flag, on a 2.66GHz Intel(R) Core(TM)2 Quad CPU.

```
interface Pattern {
    boolean start(Role role);
    boolean stop(Role role);
    boolean match(Accounting accounting);
    Accounting makeAccounting();

    interface Accounting {
        void accountFor(GraphNode node,
                Role roleOfNode);
        int overhead();
    }
}

interface PatternOccurrence {
    void incrOverhead(int bytes)
    void incrCount();
}

interface Region extends Map<Pattern,
      PatternOccurrence> {}

interface ContainerOrContainedModel {
    Role role(GraphNode node);
    Region equivClass(GraphNode node);

    enum Role { ... }; // see Table 4
}

Set<PatternOccurrences>
      computePatternOccurrences(Pattern
      pattern, Graph dataStructure,
      ContainerOrContainedModel CoC) {
    Set<PatternOccurrences> occurrences; //
          the output

    dataStructure.dfsTraverse(new GraphData.
          Visitor() {
        Stack<Accounting> accountingStack;
        boolean active;

        void preorder(GraphNode node) {
            Role role = CoC.role(node)
            if (pattern.stop(role) {
                if (!patternStack.isEmpty()) {
                    accountingStack.top().
                            accountFor(node, role
                            );
                    active = false;
                }
            } else if (pattern.start(role)) {
                active = true;
                patternStack.push(pattern.
                        makeAccounting());
            }
            if (active) {
                accountingStack.top().
                        accountFor(node, role)
            }
        }
        void postorder(GraphNode node) {
            if (pattern.start(node)) {
                region = regionStack.pop();
                if (pattern.match(
                        accountingStack.pop()) {
                    Region region = CoC.
                            equivClass(node);
                    PatternOccurrence occ =
                            region.get(pattern);
                    occ.incrCount();
                    occ.incrOverhead(
                            accountingState.
                            overhead());
                    occurrences.add(occ);
                }
            } else if (pattern.stop(node)) {
                active = true;
            }
        }
    });
    return occurrences;
}
```

Fig. 7. The algorithm that, given a pattern and a data structure, produces a set of aggregated pattern occurrences

5 Experiences with Our Tool

We have implemented the algorithm described in Sect. 4 and the eleven patterns in a tool that analyzes a heap snapshot for problematic uses of memory. It is in initial use by system test teams within IBM. The tool presents a list of the pattern occurrences, ranked by the amount of overhead they explain, and grouped by ContainerOrContained region. We have an initial visualization tool, under development, that will present these figures on a picture of the ContainerOrContained DAG. In this section, we walk through uses of the tool on nine of the heap snapshots from Fig. 1, to give a qualitative feeling of the value of the analysis, and to demonstrate the kinds of problems that routinely show up in

Table 5. The number of objects and computation time to detect all the patterns in the heap snapshots from Fig. 1.

	# objects [million]	time [minutes]		# objects [million]	time [minutes]		# objects [million]	time [minutes]
S32	102	12.56	S27	14	1.59	S11	3.76	0.13
S4	58	114.37	S13	13	100.63	S25	3.09	7.98
S8	57	1.96	S18	12	3.71	S34	2.03	0.64
S3	50	13.59	S6	11	7.25	S24	1.82	8.78
S16	49	5.68	S1	11	1.52	S5	1.73	0.14
S17	37	19.8	S21	10	22.12	S29	1.41	0.73
S2	37	8.2	S23	8.29	6.06	S12	1.41	0.51
S26	36	9.75	S7	7.77	45.86	S30	1.37	3.14
S14	34	275.62	S15	5.83	77.21	S22	1.14	1.49
S9	34	1.27	S28	5.26	0.91	S20	0.62	1.27
S19	30	12.36	S33	4.36	2.84			
S10	26	2.21	S31	4.29	73.62			

real applications. Each of these is a real application, not a benchmark. Some are servers, and some are client applications.[6]

Table 6 provides a high level overview of footprint problems detected by the tool system. Each row of the table is a row of the output of the tool: the region in which the problem occurs, the problematic pattern, and the overhead that would be saved by fixing that pattern. The next section shows that the number of rows that user must inspect is typically low; in the worst case of the snapshots we document, the user must inspect 20 rows.

For the purposes of this section, we have selected some example pattern occurrences that would be particularly easy to fix. In a few cases, you will see that this may not cover a very large fraction of the total overhead; these are the cases where it would be necessary to fix a dozen, rather than a handful of occurrences. Still, even in these cases, fixing a few lines of code can reduce memory footprint by almost 100MB — not bad for a few minutes' work. We now step through some of these cases in more detail.

Application S7. This application has a heap size of 652MB of which 517MB is overhead. The application suffers from three easy to fix problems in three collections. As shown in the S7 rows of Table 6, these belong to the sparse, small, and fixed-size collection patterns. One of the collections, a `HashMap` suffers simultaneously from both the sparse (P4) and the small (P3) collection patterns. The small collections are likely to be sparse, too. The tool has split out the costs of these two separate problems, so that the user can gauge the benefit of tackling these problems, one at a time: these two problems explain 92MB and 73MB of overhead, respectively.

The tool also specifies the remedies available for each pattern. For example, the small sparse `HashMap`s can be remedied by passing an appropriate number to the constructor. In addition to reporting the occurrence's overhead (as shown in each row of Table 6), the tool (not shown) also reports the occurrence *count*, and

[6] Their names are, unfortunately, confidential.

Table 6. Each row, a pattern occurrence, was selected as a case that would be easy for the developers to fix. As Sect. 6 shows, the developer needn't ever fix more than 20 separate problems (and often far fewer than that) to address overhead issues.

	Total Heap Size	Total Overhead(TO)	Region Pattern	Occurrence Overhead	Total Overhead Explained [Size]	[% of TO]
S7	652MB	517MB	`HashMap` P4: sparse collections	92MB	206MB	40
			`HashMap` P3: small collections	73MB		
			`HashMap` P3: small collections	19MB		
			`HashMap` P2: fixed-size collections	22MB		
S15	1.4GB	971MB	`HashSet` P1: empty collections	50MB	85MB	9
			`LinkedList` P1: empty collections	6.9MB		
			`ArrayList` P1: empty collections	6.9MB		
			`sPropContainer` P6: boxed scalar collections	21MB		
S3	2.61GB	2.26GB	`HashMap` P6: boxed scalar collections	306MB	1.1GB	49
			`SparseNode` P8: highly delegated	19MB		
			`HashMap` P6: boxed scalar collections	267MB		
			`SparseNode` P8: highly delegated	25MB		
			`UnmodifiableMap` P7: wrapped collections	108MB		
			`ConcurrentHashMap` P1: empty collections	99MB		
S8	1.28GB	1GB	`TreeMap` P6: boxed scalar collections	742MB	806MB	79
			`TreeMap` P6: boxed scalar collections	64MB		
S32	9GB	4GB	`ArrayList` P4: sparse collections	736MB	861MB	21
			`ObjArray` P4: sparse collections	72MB		
			`RedirectHashMap` P4: sparse collections	34MB		
			`ArrayList` P3: small collections	19MB		
S27	506MB	307MB	`HashSet` P1: empty collections	16MB	22MB	7
			`HttpRequestMI` P3: small collections	6.02MB		
S17	1.872GB	1.21GB	`ArrayList` P4: sparse collections	53MB	84MB	7
			`BigDecimal` P10: sparse references	18MB		
			`Date` P10: sparse references	13MB		
S29	832MB	452MB	Vector P3: small collections	107MB	143MB	32
			Vector P4: sparse collections	21MB		
			`Vector` P1: empty collections	15MB		
S4	2.87GB	2.47GB	`HashMap` P9: nested collections	422MB	422MB	17

a distribution of the sizes of the collection instances that map to that pattern occurrence. This data can be helpful in choosing a solution. For example, if 90% of the `HashMap`s that map to that occurrence have only 6 entries, then this, plus a small amount of slop, is a good figure to pass to the `HashMap` constructor. For now, the tool gives these numbers, and a set of known general solutions to the pattern of each occurrence. Though this arithmetic work necessary to gauge the right solution, is straightforward, we feel that it is something the tool should do. Work is in progress to do this arithmetic automatically.

Application S3. The application uses 362MB on actual data and 2.26GB on overhead. This application suffers from several *boxed scalar collection* pattern occurrences in `HashMap`s, accounting for 573MB of overhead. There are easy to use, off the shelf solutions to this problem, including those from the Apache Commons [1] and GNU Trove [5] libraries.

The tool also finds a large occurrence of a *wrapped collections* pattern. This region is headed by a collection of type `Collections$UnmodifiableMap`; in the Java standard libraries, this is a type that wraps around a given `Map`, changing its accessibility characteristics. The tool (not shown) reveals an occurrence count of 2,030,732 that accounts for 108MB of overhead. The trivial solution to this problem is to avoid, at deployment time, the use of the unmodifiable wrapper.

Application S8. The application consumes 1.28GB, of which 1GB is overhead. As with Application S3, the main contributors to the overhead are occurrences of *boxed scalar collection* pattern. In this case, the guilty collections are two `TreeMaps`; one is especially problematic, being responsible for 742MB of overhead. Upon consultation with the developers, we learned that this application does not need the sorted property of the map until the map is fully populated. A solution that stores the map as parallel arrays of the primitive data, and sorts at the end would eliminate this overhead entirely — thus saving 1GB of memory.

Application S32. This application has a memory footprint of 9GB, of which 4GB is overhead. The main findings belong to the *sparse collection* pattern. The largest occurrence is an `ArrayList` region that consumes 1.43GB (not shown), and 736MB of these lists are used by empty array slots.

Application S4. The application spends 407MB on actual data and 2.47GB on overhead. The tool's main finding is a *Hash Map of ArrayList* pattern which accounts for 422MB of overhead. In this case, the single outer map had many inner, but relatively small, lists. Though not small enough to fire the *small collections* pattern, this case fires the nested collections pattern. In general for this pattern, if the outer collection has a significant larger number of elements than the inner collection, the memory overhead may be reduced by switching the order of collection's nesting. The benefit comes as a consequence of greatly reducing the total number of collection instances.

6 Validation and Characterization

The detection of costly memory patterns provides support for understanding how a particular system uses (or misuses) its memory. In this section we look at the results of our analysis across a range of real-world applications, with two goals in mind. First, we aim to validate that the approach, along with the particular patterns, produces effective results. Second, we employ the analysis to shed light on how Java programmers introduce inefficiencies in their implementations. This characterization can help the community better determine what sort of additional tools, optimizations, or language features might lead to more efficient memory usage. There are three broad questions we would like to assess: 1) Do the patterns we have identified explain a significant amount of the overhead? 2) How often do these patterns occur in applications? and 3) Do the patterns provide the best candidates for memory footprint reduction? To achieve this we

introduce a set of metrics, and apply them on a set of fifteen real-world applications. In summary, applying the metrics on the application set shows that the memory patterns we detect do indeed explain large sources of overhead in each application.

In Fig. 1 we present examples of applications with large footprint overhead. For our study, we select from these the fifteen applications which make the least efficient use of memory, applications where more than 50% of the heap is overhead. Note that in the reporting of results, we are careful to distinguish between a *pattern category*, such as *Empty Collections*, and a *pattern occurrence*, an instantiation of a pattern category at a particular context. In all of the computations of overhead explained by the tool, we consider only pattern occurrences which account for at least 1% of the application overhead. We choose this threshold so that we report and ask the user to remediate only nontrivial issues. We do not want to burden the user with lots of insignificant findings (i.e. of just a few bytes or kilobytes). To ensure meaningful comparisons, the computation of total overhead in the heap is based on traversing the entire heap and tabulating lower-level overheads like object header costs, as described in 3.4 (i.e. it is not dependent on pattern detection). Thus it includes all sources of overhead, from trivial and nontrivial cases alike.

6.1 How Much of the Overhead Do the Patterns Explain?

An important measure of the effectiveness of the analysis approach, and of the particular patterns we have identified, is whether they explain a significant portion of the memory overhead. We introduce an *overhead coverage* metric to quantify this.

Definition 4. Overhead coverage *measures the percentage of total memory overhead explained by the memory patterns detected.*

Table 1 gives a summary of the coverage across all the heaps we analyzed (not just the subset of fifteen with high overhead). In almost half of the heaps the tool is able to explain more than 80% of the overhead. Fig. 8 gives a more detailed look at the fifteen highest overhead heaps. The third bar shows us the percentage of overhead explained by 100% of the discovered pattern occurrences.

The remaining, unaccounted for overhead can be useful as well in helping us identify important missing patterns. In continuing work we are looking at the unexplained part of the heap, and are identifying which portion is the result of detecting trivial occurrences of known patterns, and which are new patterns that need to be encoded.

6.2 How Many Contexts Does a User Need to Look at?

We would like to assess if an application which contains memory inefficiencies has its problems scattered around the entire application or if the main sources of overhead are located in just a few contexts. As one validation of the usefulness

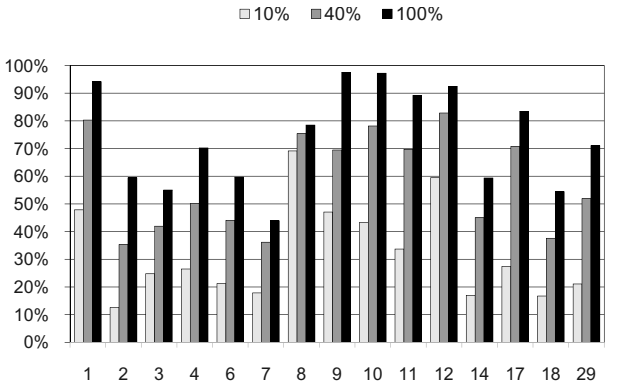
of the pattern detection tool, this can tell us whether a user can examine a manageable number of reported contexts and have a significant impact on memory footprint. From the tool perspective, the metric provides understanding into the depth of the patterns occurrences that need to be reported in order for the tool to be useful. This result is also useful from a characterization standpoint, since it can suggest whether problems are better addressed by automated optimizations (e.g. if they are diffuse) rather than by tools (e.g. if they are concentrated). The tool reports pattern occurrences ordered by the amount of overhead they explain. We would like to know how significant a part of the memory overhead is explained by the first reported findings. To achieve this we introduce the *Pattern occurrence concentration metric.*

Definition 5. Pattern occurrence concentration *shows the percentage of the heap overhead which is explained by the top N% of the memory pattern occurrences detected.*

Fig. 8 presents the results of the pattern occurrence concentration metric applied to the experimental set of applications. The figure reveals the percentage of the heap overhead which is explained by N% of the top memory pattern occurrences detected, where N is 10%, 40% and 100%. We see that on average more than 25% of the overhead is explained by the top 10% of the findings. In particular Application 8 has almost 70% of the overhead explained by the top 10% of pattern occurrences. This 70% of overhead is uncovered by one single pattern occurrence, as seen in Fig. 8(b). This means that the user has to apply only one fix to greatly reduce the application's overhead footprint. Having a single context explain the bulk of the overhead is not typical; usually there are multiple contexts that need to be addressed. From the results we can also see that increasing the percentage of pattern occurrences from 40% to 100% does not provide a significant increment in the total overhead explained, compared to the number of patterns needed to explain that difference. This means that the applications have a few important memory issues and the rest are smaller findings. For instance, in the case of Application 4 the top 40% and 100% of occurrences explain 50% and 70% of the overhead respectively. 12 pattern occurrences are required to uncover that additional 20% of overhead. Thus we can conclude that the main contributors to the memory overhead are exposed by a few top occurrences.

6.3 How Many Different Types of Problems Exist in a Single Heap?

The previous metric offers quantitative information about how much of the heap overhead is explained by the top pattern occurrences. Next, we want to identify whether an application suffers from a wide range of memory issues or if it contains only a few types of problems. Moreover, as in the previous metric, we would like to understand how many pattern categories are needed to account for a significant portion of the overhead in a given heap.



(a) The percentage of total overhead explained by top N% of pattern occurrences

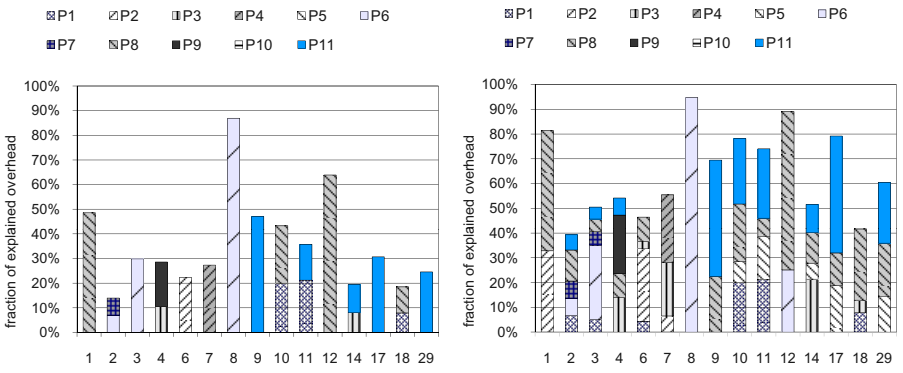
	1	2	3	4	6	7	8	9	10	11	12	14	17	18	29
10%	1	2	2	2	2	1	1	1	2	2	1	2	1	2	1
40%	2	6	6	8	8	3	2	2	5	6	2	8	4	8	4
100%	4	14	13	20	18	6	3	3	11	13	4	18	8	19	10

(b) Number of pattern occurrences in the top N%

Fig. 8. Occurrence Concentration

Definition 6. Pattern category concentration *is the percentage of the heap overhead which is explained by the pattern categories represented in the top N% of memory pattern occurrences detected.*

Fig. 9(a) and Fig. 9(b) depicts the category concentration results using the top 10% and 40% of findings respectively. The results show that there are always only a few categories of memory issues responsible for most of the overhead. Application 8 exhibit one type of memory inefficiency and most of the applications have two or three major contributors to the memory overhead. If we compare the pattern categories reported in both cases, for N=10% and N=40%, we note that there are no major new types of problems detected by increasing the number of pattern occurrences, even when more overhead is explained. Table 7 gives a numerical summary of the data. We can see that usually a system suffers from a small number of types of memory inefficiency, for N=40% there are 3 different issues. We can also observe that the same issue is seen in several different points in the application. This may be a consequence of the degree of reuse common in framework-based applications, or of developers coding with a similar style in multiple places in the code.



(a) The percentage of overhead explained by the pattern categories represented in the top N=10% of pattern occurrences

(b) The percentage of overhead explained by the pattern categories represented in the top N=40% of pattern occurrences

Fig. 9. Category Concentration

6.4 How often Do the Pattern Categories Appear across Applications?

We have seen that the top findings reveal the main categories of memory inefficiencies usage in a given application. The next question is how often are the same patterns seen across different applications? Does a small number of pattern categories explain the bulk of the overhead in most applications, or is there more variety among applications? To study this across multiple systems we introduce a pairwise similarity metric.

Definition 7. Pattern category similarity *measures the degree to which two applications contain the same pattern categories. The similarity metric reports the ratio of the number of pattern categories common to both applications to the total number of pattern categories detected in the two applications:*

$$CS = \frac{2|PC_1 \bigcap PC_2|}{|PC_1| + |PC_2|}$$

where PC is the set of pattern categories detected in the applications 1 and 2.

The value of pattern category similarity metric belongs to [0, 1]. A value of 1 means that the same pattern categories have been detected in both applications. The lower the value of the similarity metric the greater the range of problems identified across two applications.

Fig. 10 reports the similarity metric, computed pairwise for the heaps in our experimental set. The darker the gray shade, the more common the problems detected between the two applications. To understand how a given heap compares to each of the other heaps, look at both the row and column labeled with

Table 7. Number of pattern categories represented in the top 10% and the top 40% of pattern occurrences

	# categories		
	min	median	max
top 10% of occurrences	1	1	2
top 40% of occurrences	1	3	5

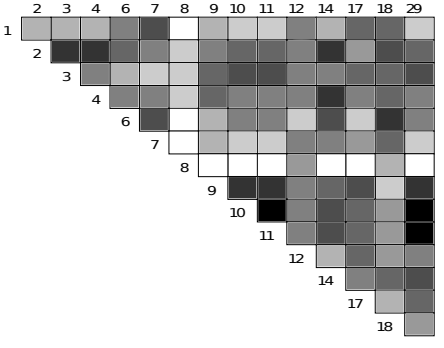


Fig. 10. Pattern category similarity. The darker the gray shade, the more common are the pattern categories found in the two applications.

the given heap (i.e. an L-shaped region of cells). There is no single application that presents completely different pattern categories compared with all the other applications, though application 8 is the least similar to the others. Eleven applications out of fifteen contain half of the categories in common with at least 9 applications (i.e. CS$\geq$0.5). From the results we conclude that the same memory problems are frequently seen across multiple applications.

6.5 Additional Sources of Inefficiency

The current work addresses patterns of memory usage that have a high representational overhead, using a definition of overhead based on infrastructure costs such as object headers and collection implementations. Developers also introduce inefficiencies in the way they represent the data proper. For example, in our experience we have seen many applications with large amounts of duplicate immutable data, scalar data such as enumerated types that are represented in text form, or classes carrying the cost of unused fields from overly general base classes. In future work we would like to address these inefficiencies by encoding data patterns into the same framework. Much of the existing analysis approach can be easily extended to support recognition of these patterns.

7 Related Work

We discuss related work in two main categories.

Patterns. One other recent work has adopted a similar approach to memory footprint issues with a focus on collections [16]. That paper presents a solution based on a language for specifying queries of heap structure. We have found that their language, and the system as a whole, is insufficient for expressing important patterns. They hard-code collection types, and do not treat any issues outside the scope of those collections; e.g. there is no discussion of delegation or sparse references for the items stored within collections. Their aggregation is based on allocation context of *individual* objects, with, in some cases, a hard-coded "k" of context in order to cope with the implementation details of the common collection classes.

Memory Tools. Many tools put an emphasis on detecting memory leaks [12,7,17,14], rather than footprint. Several tools place an emphasis on *dominance* [3,18,10], and a great many tools [2,7,3,18,17,16] use the raw data types of the program as the basis for aggregation and reporting. Some works focus on the visualization side of memory analysis. They attempt, through cleverly designed views, to allow the human to make sense of the heap [2,6], or JVM-level behaviors [15]. These are indispensable tools for experts, but our experience shows these to be of less value among the rest of the development teams.

8 Conclusions

In Java, it is difficult to implement a data model with only a single data type. The language's lack of support for composition and unions forces us to delegate functionality to side objects. The sometimes perverse focus on reuseability and pluggability in our frameworks [11] encourages us to to favor delegation over subclassing [4,8]. For these reasons, classes are a low level manifestation of intent. In Java, even the most basic of data types, the string, requires two types of objects and delegation: a `String` pointing to a character array.

There is a wide *modeling gap* between what programmers intend to represent, and the ways that the language and runtime encourage or force them to store this information. As a consequence, most Java heaps suffer from excessive implementation overhead. We have shown that it is possible to identify a small set of semantic reasons for the majority of these overheads in Java heaps. In the future, we would like to explore this modeling gap more thoroughly. It is possible that a more rigorous study of the gap will yield opportunities close it, for important common cases. Why must we use collections explicitly, to express concerns that are so highly stylized: relationships, long-lived repositories, and transient views?

Acknowledgements. Adriana E. Chis is funded by an IRCSET/IBM Enterprise Partnership Scheme Postgraduate Research Scholarship.

References

1. Apache: Commons library, http://commons.apache.org
2. De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., Yang, J.: Visualizing the Execution of Java Programs. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 151–162. Springer, Heidelberg (2002)
3. Eclipse Project: Eclipse memory analyzer, http://www.eclipse.org/mat/
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
5. GNU: Trove library, http://trove4j.sourceforge.net
6. Hill, T., Noble, J., Potter, J.: Scalable visualizations of object-oriented systems with ownership trees. J. Vis. Lang. Comput. 13(3), 319–339 (2002)
7. Jump, M., McKinley, K.S.: Cork: dynamic memory leak detection for garbage-collected languages. In: Symposium on Principles of Programming Languages (2007)
8. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in a class-based object-oriented programming language. In: International Conference on Software Engineering (2008)
9. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1(1), 121–141 (1979)
10. Mitchell, N., Schonberg, E., Sevitsky, G.: Making sense of large heaps. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 77–97. Springer, Heidelberg (2009)
11. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to java runtime bloat. IEEE Software 27, 56–63 (2010)
12. Mitchell, N., Sevitsky, G.: Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003)
13. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: Object-oriented Programming, Systems, Languages, and Applications, pp. 245–260. ACM, New York (2007)
14. Novark, G., Berger, E.D., Zorn, B.G.: Efficiently and precisely locating memory leaks and bloat. In: Programming Language Design and Implementation, pp. 397–407. ACM, New York (2009)
15. Printezis, T., Jones, R.: Gcspy: an adaptable heap visualisation framework. In: Object-Oriented Programming, Systems, Languages, and Applications, pp. 343–358. ACM, New York (2002)
16. Shacham, O., Vechev, M., Yahav, E.: Chameleon: adaptive selection of collections. In: Programming Language Design and Implementation, pp. 408–418. ACM, New York (2009)
17. Xu, G., Rountev, A.: Precise memory leak detection for java software using container profiling. In: International Conference on Software Engineering, pp. 151–160. ACM, New York (2008)
18. Yourkit LLC: Yourkit profiler, http://www.yourkit.com