Detecting Inefficiently-Used Containers to Avoid Bloat

Guoqing Xu   Atanas Rountev
Ohio State University
{xug,rountev}@cse.ohio-state.edu

Abstract

Runtime bloat degrades significantly the performance and scalability of software systems. An important source of bloat is the inefficient use of containers. It is expensive to create inefficiently-used containers and to invoke their associated methods, as this may ultimately execute large volumes of code, with call stacks dozens deep, and allocate many temporary objects.

This paper presents practical static and dynamic tools that can find inappropriate use of containers in Java programs. At the core of these tools is a base static analysis that identifies, for each container, the objects that are added to this container and the key statements (i.e., heap loads and stores) that achieve the semantics of common container operations such as *ADD* and *GET*. The static tool finds problematic uses of containers by considering the nesting relationships among the loops where these *semantics-achieving statements* are located, while the dynamic tool can instrument these statements and find inefficiencies by profiling their execution frequencies.

The high precision of the base analysis is achieved by taking advantage of a context-free language (CFL)-reachability formulation of points-to analysis and by accounting for container-specific properties. It is demand-driven and client-driven, facilitating refinement specific to each queried container object and increasing scalability. The tools built with the help of this analysis can be used both to avoid the creation of container-related performance problems early during development, and to help with diagnosis when problems are observed during tuning. Our experimental results show that the static tool has a low false positive rate and produces more relevant information than its dynamic counterpart. Further case studies suggest that significant optimization opportunities can be found by focusing on statically-identified containers for which high allocation frequency is observed at run time.

Categories and Subject Descriptors F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; D.3.4 [*Programming Languages*]: Processors—Memory management, optimization; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Algorithms, Languages, Performance

Keywords Container bloat, CFL reachability, points-to analysis

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06. . . $10.00

1. Introduction

While Java developers need not worry about memory correctness issues, it remains challenging for them to efficiently use memory. As a result, memory bloat [22, 23, 26, 35, 42] commonly exists in large-scale Java applications. Evidence has been found that bloat has significant impact on performance, leading to reduced scalability and usability [22].

The inefficient use of containers is an important source of systemic bloat. Programming languages such as Java include a collection framework which provides abstract data types for representing groups of related data objects (e.g., lists, sets, and maps). Based on this collection framework, one can easily construct application-specific container types such as trees and graphs. Real-world programs make extensive use of containers, both through collection classes and through user-defined container types. Programmers allocate containers in thousands of code locations, using them in a variety of ways, such as storing data, implementing unsupported language features such as returning multiple values, wrapping data in APIs to provide general service for multiple clients, etc.

Containers are easy to misuse. A memory leak could occur if objects added to a container are not removed after they are no longer used [3, 4, 15, 44]. Another problematic situation is when a container, while not strictly necessary for what it is supposed to accomplish, is used nevertheless. Although most such containers are eventually garbage-collected and allocating them may not lead to `OutOfMemory` errors, they can have conspicuous impact on performance. For example, as reported in [42], finding and specializing inefficiently-used HashMap can reduce the number of allocated objects in a server application by more than half. Identifying containers that consume excessive resources (for what they accomplish) is an important step toward finding potential container-related optimization opportunities.

Motivation and problems The extensive use of containers in Java software makes it impossible to manually inspect the choice and the use of each container in the program. Dynamic analysis has typically been the weapon of choice for detecting memory leaks and bloat in real-world Java software. A dynamic bloat detector finds performance problems using a "from-symptom-to-cause" approach: it starts by observing suspicious behaviors and then attempts to locate the cause from the observed symptom. One fundamental problem for such approaches is the selection of appropriate symptoms. It can be extremely difficult to define symptoms that can precisely capture the target bloat—in many cases, the suspicious behaviors that the tool intends to capture are not unique characteristics for the problematic program entities. Very often, they can also be observed from entities that function appropriately, leading to false positives and imprecise reports. The first research problem investigated in this paper is *whether it is possible to alleviate this limitation of purely dynamic bloat analyses by using a static or a hybrid technique.* A static (or a hybrid static/dynamic) technique

```
class CUP$LexParse$actions {
    RegExp makeNL(){
        Vector list = new Vector();
        list.addElement(new Interval(’\n’,’\r’));
        list.addElement(new Interval(’\u0085’,’\u0085’));
        list.addElement(new Interval(’\u2028’,’\u2029’));
        RegExp1 c = new RegExp1(sym.CCLASS, list);
        ...
    }
}
                        (a)
```

```
class Httpd extends HttpConnection{
  Reply recvReply(Request request){
    if (request.getPath().equals("/admin/enable")){
      Hashtable attrs = cgi(request);
      String config = (String) attrs.get("config");
      String filter = (String) attrs.get("filter");
      ...
    }else if(request.getPath().equals("/admin/createConfig")){
      Hashtable attrs = cgi(request);
      String config = (String) attrs.get("config");
      ...
    } ...
  }
  Hashtable cgi(Request request){
    Hashtable attrs = new Hashtable(13);
    String query = request.getQueryString();
    String data = request.getData();
    if (query != null){
      StringTokenizer st = new StringTokenizer(decode(query), "&");
      while (st.hasMoreTokens()){
         String token = st.nextToken();
         String key = token.substring(0, token.indexOf(’=’));
         String value = token.substring(token.indexOf(’=’) + 1);
         attrs.put(key, value);
      }
    } ...
    return attrs;
  }
}
                         (b)
```

Figure 1. (a) An example of an *underutilized container*, extracted from program *JFlex*, a scanner generator for Java. `Vector` *list* is used only to pass the three `Interval` objects to the constructor of `RegExpr1`; (b) An example of an *overpopulated container*, extracted from program *Muffin*, a WWW filtering system written in Java. Given a request, method `cgi` always decodes the entire request string into `Hashtable` *attrs*. However, this `HashMap` is later subjected to at most two lookup operations.

may be able to reduce false positives by exploiting certain program properties inherent in the source code.

The second major problem addressed in the paper is related to understanding the semantics of containers. Of existing dynamic bloat detection techniques, tools from [44] and [35] were designed specifically for finding container-related problems. Both tools require the user to provide annotations for each container type, which are subsequently used to understand the container semantics and relate container behavior to the profiling frameworks. However, it may be a heavy burden for the programmer to complete these annotations. In addition, when the interface of a container changes, its annotations have to be revised as well. Creating such annotations is also impractical when the container types come from large libraries and frameworks developed by others (e.g., containers of transaction data in enterprise applications). To reduce such an annotation burden and make the container analysis more general, the paper explores the possibility of *extracting the container semantics automatically from the container implementation, with minimal need for user interaction.*

Targeted container inefficiencies We aim to detect containers that are inefficiently used in the following two ways:

- ***Underutilized container.*** A container is *underutilized* if it holds a very small number of elements during its lifetime. It is wasteful to use an underutilized container to hold data. First, a container is usually created with a default number of slots (e.g., 16), and a big portion of the memory space is wasted if only a few slots are occupied. If the size of the container is fixed (e.g., 1), a specialized container type such as `Collections.singletonSet` can be employed to replace the original general type (e.g., `HashSet`). Second, the functionality associated with the container type may be much more general than what is actually needed. For example, the process of retrieving an element from a `HashSet` involves dozens of calls. If there is a small number of objects in the `HashSet`, it may be possible for a performance expert to replace this `HashSet` by introducing extra local variables, parameters, or an array.

  Figure 1(a) illustrates an example of this type of problem, which was reported by our tool. The example is extracted from *JFlex*, a scanner generator from our benchmark set. Method `makeNL` creates a `Vector` object that by default allocates a 10-element array. The only purpose of this object is to pass the three `Interval` objects into the constructor of `RegExpr1`. This `Vector` object is allocated more than 10,000 times during the execution of a large test case. In fact, there are many locations in the code where `Vector` objects are created solely for this purpose. Creating specialized constructors of `RegExpr1` that allow the direct passing of `Interval` objects can avoid the allocation/deallocation of thousands of objects.

- ***Overpopulated container.*** This problem occurs if there is a container that, while holding many objects, is looked up only a few times. Due to unnecessary data elements, memory is wasted and it takes longer to perform a container operation. In this case, a programmer may be able to inspect the code to find which objects will definitely *not* be retrieved from the container, and then find a way to avoid adding these objects or even creating them (if they are never used). Figure 1(b) shows an example of this problem, which was found by our analysis in *Muffin*, a WWW filtering system from our benchmark set. Many strings are generated and added into a `Hashtable`, but only the entries with keys `"config"` and `"filter"` are eventually requested. Instead of decoding and bookkeeping the entire request string, a specialized version of method `cgi` could declare an additional string parameter representing the requested key, and return the corresponding value immediately when the given key is found.

Base static analysis: extracting container semantics There are three major technical challenges in automatically extracting semantics for different container types and implementations. The first challenge is to establish a unified model for different container types. For example, consider two concrete container classes in Java, `Hashtable` and `LinkedList`. The unified model has to capture the common behaviors that characterize their "container" property while ignoring the differences in their specific implementations and usage domains. To address this challenge, we propose to treat all container classes as an abstract data type with two basic operations: *ADD* and *GET*. Consider `Hashtable` and `LinkedList` again: despite their many differences, we are interested only in the process by which objects are added to (*ADD*) and retrieved from (*GET*) these containers. Details of their implementations and usage (e.g., whether they store key-value pairs or individual objects) are abstracted away.

The second challenge is to select program entities that correspond to these abstract operations. Research from [35, 44] focuses on methods. For example, methods `put` and `get` implement the semantics of *ADD* and *GET*, respectively, for class `Hashtable`. While identifying abstract operations at the method level is a straightforward idea, it is impossible to perform without user annotations because different container types use different methods for these operations. To automate this process, we propose to operate at the statement level. The key insight is that the core behavior of each operation can be implemented by a single statement. The statements that implement the *ADD* and *GET* operations are usually heap stores and loads, respectively. Such statements will be referred to *semantics-achieving* statements. For example, for class `ArrayList`, the statement achieving the functionality of *ADD* is a heap store `array[i] = o` in method `ArrayList.add`, where `array` refers to the backing array of the list and `o` is a formal parameter referring to the object[1] to be added. Identifying semantics-achieving statements bridges the gap between the low-level code analysis and the high-level container semantics.

The third challenge is to develop precise and efficient algorithms to discover container structures. The identification of semantics-achieving statements for a particular container object requires reasoning about the container data structure, in order to detect the objects that are added to the container from the client code (i.e., *element objects*) as well as the helper objects that are created by the container (i.e., *inside objects*). There are usually multiple layers in the data structure of a container type, and a naive approach based on points-to analysis may not be able to distinguish among elements added to different container objects that have the same type. To obtain precise information about the use of a container, it is crucial to prune, context-sensitively, nodes and edges irrelevant to the container in the points-to graph. While there exists a body of precise reasoning techniques such as shape analysis (e.g., [6, 7, 34]) and decision procedures for verification of pointer-based data structures (e.g., [17, 20]), these analyses tend to be expensive and do not scale well to large applications.

Our analysis attempts to refine the object sub-graph rooted at each container object by taking advantage of the CFL-reachability formulation of pointer aliasing. The key observation is that if an object o can be reached from a container object c through (direct or transitive) field dereferences, there must exist a chain of stores of the form $a_0.f_0 = o; a_1.f_1 = b_0; a_2.f_2 = b_1; \ldots; a_n.f_n = b_{n-1}; b_n = c$, such that the two reference variables in each pair (a_i, b_i) for $0 \leq i \leq n$ are aliases. Because aliasing relationships can be computed by solving CFL-reachability on a flow graph [37], the goal of our analysis is to find valid paths (in terms of both heap accesses and method calls) on the flow graph that contain such chains of stores. We consider all objects o that have such paths reaching the container object and that are not inside objects created by the container. Among those, element objects are the ones that have a chain of stores $a_0.f_0 = o; a_1.f_1 = b_0; \ldots$ such that all a_i and b_i along the chain point to inside objects of the container. We have successfully applied this demand-driven analysis to large Java applications, including the `eclipse` framework (and its plugins). The description of the analysis can be found in Section 2.

Static inference and dynamic profiling of execution frequencies If the *ADD* operations of a container are executed a very small number of times, the container may suffer from an underutilization problem. If the frequency of its *GET* operations is much smaller than the frequency of its *ADD* operations, the container may be overpopulated. The next step of the analysis is to compare the frequencies of these operations, using the semantics-achieving statements (annotated with the relevant calling contexts) identified by the base analysis.

A natural choice for comparing the frequencies of the semantics-achieving statements is to instrument these statements and to develop a dynamic analysis by profiling the *observed* frequencies. However, the usefulness of this approach may be limited because it does not directly point to the underlying cause of the problem. Furthermore, the generated results depend completely on the specific inputs and runs being observed: containers whose behaviors are suspicious in one particular run may behave appropriately in other runs, making it hard to identify problematic containers.

An alternative is to design a static analysis that detects performance problems by looking for certain source code properties that can *approximate* the relationship between execution frequencies, regardless of inputs and runs. There exist a number of analyses that can be employed to infer such a relationship. For example, semantics-achieving statements are often nested in loops. Various techniques such as interval analysis [40] and symbolic bound analysis [11, 12] may be used to discover the loop bounds. However, such techniques are often ineffective in handling dynamic heap data structures, and it is difficult to scale them to large programs. We take a much simpler approach where data flow does not need to be considered: relative frequencies are inferred based on the nesting of the loops where the semantics-achieving statements are located. Despite this simplicity, the inferred relationships are execution independent and, in our experience, lead to low false positive rates when used to find optimizable containers.

We have implemented both the dynamic frequency profiler (Section 3.1) and the static inference analysis (Section 3.2). Detailed comparison between them is provided in Section 3.2 and demonstrated experimentally in Section 4.

Features of the base analysis The base analysis needs to be sufficiently precise, as both the static inference algorithm and the dynamic frequency profiler rely on it to find problematic containers. Our algorithmic design is focused on three important features of the analysis. First, since we are interested only in containers, the algorithm is *demand-driven*, so that it can perform only the work necessary to answer queries about the usage of containers. Second, because a container type can be instantiated many times in the program, failure to distinguish elements added into different container objects of the same type could result in a large number of false positives. To avoid this, if the analysis cannot find a highly-precise solution under a client-defined time budget, it does *not* report any *ADD* and *GET* operations (instead of reporting them based on over-conservative approximations). In a practical tool that identifies potential bloat, the precise identification of inefficiently-used containers (i.e., reducing the false positive rate) is much more important than reporting all potentially problematic ones with many false warnings (i.e., reducing the false negative rate). This choice aims at higher programmer productivity and real-world usefulness. Finally, the analysis is *client-driven*, as the amount of information it produces can be controlled by the client-defined time budget.

Evaluation Section 4 presents experimental results showing that the static tool successfully finds inefficient uses of containers. It generates a total of 295 warnings for the 21 Java programs in our benchmark set. For each benchmark, we randomly picked 20 warnings for manual inspection. Among those, we found a very small number of false positives (e.g., 4 for the largest benchmark `eclipse`). Further experiments showed that (1) most of the statically reported containers indeed exhibit problematic behaviors at run time, and (2) the inefficient uses of these containers are much easier to understand than the uses of containers reported by the dynamic analysis.

The static inference approach is useful for detecting container problems during coding, before performance tuning has started. It is a good programming practice to fix (static) performance warn-

[1] From now on we will use "object" to denote the static abstraction (i.e., allocation site) of a set of run-time objects created by the allocation site.

```
class ContainerClient{
 static void main(String[] args){
    ContainerClient client = new ContainerClient();
    Container c = new Container();
    for(int i = 0; i < 1000; i++){
      client.addElement(c, new Integer(i));
    }
    client.foo(c);
    client.bar();
  }
  void foo(Container n){
    Integer i = (Integer)n.get(10);
    Container d = new Container();
    addElement(d, new String("first"));
    addElement(d, new String("second"));
    String s = (String)d.get(0);
  }
  void bar(){
    for(int j = 0; j < 5; j++){
      Container a = new Container();
      for(int i = 0; i < 10; i++)
         addElement(a, new Double(i));
      for(int i = 0; i < a.size(); i++){
         Double b = (Double)a.get(i);
          ...
      }
     }
  }
  void addElement(Container c, Object e){
    c.add(e);
  }
 }

 class Container{
  Object[] arr;
  int pos = 0;
  Container(){
    t = new Object[1000]; this.arr = t;
  }
  void add(Object e){
    t = this.arr; t[pos++] = e;
  }
  Object get(int index){
    t = this.arr; ret = t[index]; return ret;
  }
 }
```

Figure 2. Running example.

ings early, in order to avoid potential performance problems before they pile up and become observable. It has already been recognized [22] that bloat can easily accumulate when insufficient attention is paid to performance during development. Once coding is complete and performance tuning starts, information about runtime frequency of container allocation can focus the programmer's attention on statically-identified containers that are most likely to provide optimization payoffs.[2] Using this approach, we studied the warnings for the DaCapo `bloat` and `chart` benchmarks, and easily identified fixes that reduced object creation rates by 30% for `bloat` and 5% for `chart`, leading to execution time reduction of 24.5% for `bloat` and 3.5% for `chart`. These promising initial findings suggest that our tools could be useful in practice to find and exploit opportunities for performance gains.

2. Formulation of Container Operations

This section starts with an outline of the CFL-reachability formulation of context-sensitive points-to/alias analysis for Java [37]. We formulate the base analysis for identification of semantics-achieving statements as a new CFL-reachability problem, and then present algorithms to solve this problem.

2.1 CFL-Reachability Formulation of Points-to Analysis

A variety of program analyses can be stated as CFL-reachability problems [30]. CFL-reachability is an extension of standard graph reachability that allows for filtering of uninteresting paths. Given a directed graph with labeled edges, a relation R over graph nodes can be formulated as a CFL-reachability problem by defining a context-free grammar such that a pair of nodes $(n, n') \in R$ if and only if there exists a path from n to n' for which the sequence of edge labels along the path is a word in the language L defined by the grammar. Such a path will be referred to as an L path. If there exists an L path from n to n', then n' is L-reachable from n (denoted by $n\ L\ n'$). For any non-terminal S in L's grammar, S paths and $n\ S\ n'$ are defined similarly.

Existing work on points-to analysis for Java [37, 39] employs this formulation to model (1) context sensitivity via method entries and exits, and (2) heap accesses via object field reads and writes. A demand-driven analysis is formulated as a single-source L-reachability problem which determines all nodes n' such that $n\ L\ n'$ for a given source node n. The analysis can be expressed by CFL-reachability for language $L_{\mathsf{F}} \cap R_{\mathsf{C}}$. Language L_{F}, where F stands for "flows-to", ensures precise handling of field accesses. Regular language R_{C} ensures a degree of calling context sensitivity. Both languages encode balanced-parentheses properties.

L_{F}-reachability is performed on a graph representation G of a Java program (sometimes referred to as a *flow graph*), such that if a heap object represented by the abstract location o can flow to variable v during the execution of the program, there exists an L_{F} path in G from o to v. The flow graph is constructed by creating edges for the following canonical statements: an edge $o \xrightarrow{\mathsf{new}} x$ is created for an allocation x = new O; an edge $y \xrightarrow{\mathsf{assign}} x$ is created for an assignment $x = y$; edges $y \xrightarrow{\mathsf{store(f)}} x$ and $y \xrightarrow{\mathsf{load(f)}} x$ are created for a field write $x.f = y$ and a field read $x = y.f$, respectively. Parameter passing is represented as assignments from actuals to formals; method return values are treated similarly. Writes and reads of array elements are handled by collapsing all elements into an artificial field arr_elm.

Language L_{F} Consider a simplified flow graph G with only new and assign edges. In this case the language is regular and its grammar can be written simply as $\mathit{flowsTo} \ \rightarrow\ \mathsf{new}\,(\,\mathsf{assign}\,)^*$, which shows the transitive flow due to `assign` edges. Clearly, $o\ \mathit{flowsTo}\ v$ in G means that o belongs to the points-to set of v.

For field accesses, inverse edges are introduced to allow a CFL-reachability formulation. For each graph edge $x \rightarrow y$ labeled with t, an edge $y \rightarrow x$ labeled with $\bar{t}$ is introduced. For any path p, an inverse path $\bar{p}$ can be constructed by reversing the order of edges in p and replacing each edge with its inverse. In the grammar this is captured by a new non-terminal $\overline{\mathit{flowsTo}}$ used to represent the inverse paths for $\mathit{flowsTo}$ paths. For example, if there exists a $\mathit{flowsTo}$ path from object o to variable v, there also exists a $\overline{\mathit{flowsTo}}$ path from v to o.

May-alias relationships can be modeled by defining a non-terminal alias such that $\mathit{alias} \ \rightarrow\ \overline{\mathit{flowsTo}}\,\mathit{flowsTo}$. Two variables a and b may alias if there exists an object o such that o can flow to both a and b. The field-sensitive points-to relationships can be modeled by $\mathit{flowsTo} \ \rightarrow\ \mathsf{new}\,(\,\mathsf{assign} \mid \mathsf{store(f)}\ \mathit{alias}\ \mathsf{load(f)}\,)^*$. This production checks for balanced pairs of $\mathsf{store(f)}$ and $\mathsf{load(f)}$ operations, taking into account the potential aliasing between the variables through which the store and the load occur.

Language R_{C} The context sensitivity of the analysis ensures that method entries and exits are balanced parentheses: $C \ \rightarrow\ \mathsf{entry(i)}\ C\ \mathsf{exit(i)} \mid C\ C \mid \epsilon$. Here $\mathsf{entry(i)}$ and $\mathsf{exit(i)}$ correspond to the i-th call site in the program. This production describes only a subset of the language, where all parentheses are fully balanced.

[2] Allocation frequencies can even be collected *before* the static analysis, allowing the demand-driven static algorithm to focus on hot containers.

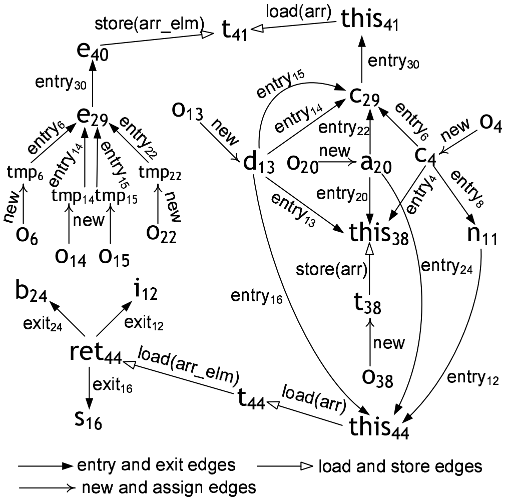

Figure 3. Flow graph for the running example.

Since a realizable path does not need to start and end in the same method, the full definition of R_{C} also allows a prefix with unbalanced closed parentheses and a suffix with unbalanced open parentheses [37]. In the absence of recursion, the balanced-parentheses language is a finite regular language (thus the notation R_{C} instead of L_{C}); approximations are introduced as necessary to handle recursive calls. Context sensitivity is achieved by considering entries and exits along a L_{F} path and ensuring that the resulting string is in R_{C}. For the purposes of this context-sensitivity check, an $\overline{\mathsf{entry}}$ edge is treated as an exit edge and vice versa.

2.2 Example

The code in Figure 2 shows an example used for illustration of the static analysis throughout the paper. The example is based on a common usage scenario of Java containers. A simple implementation of a data structure `Container` is instantiated three times (at lines 4, 13 and 20) by a client `ContainerClient`. In the example code, statement $t[pos{+}{+}] = e$ (at line 41) is the one that achieves the functionality of *ADD*, and statement $ret = t[index]$ (at line 44) is the one achieving the functionality of *GET*. Of course, these statements do not make much sense by themselves. For each semantics-achieving statement, we also need to identify the calling contexts that are relevant to the container of interest. The (statement,contexts) pair is used later to find underutilized and overpopulated containers.

Through this example, we show how the CFL-reachability formulation of points-to analysis works. We will use t_i to denote the variable t whose first occurrence is at line i, and o_i to denote the abstract object for the allocation site at line i. For example, e_{40} and o_4 represent variable e declared at line 40 and the `Container` object created at line 4. As another example, o_{14} represents the `String` object created at line 14. Node tmp_i denotes a temporary variable created artificially to connect an object and an actual parameter. For example, tmp_6 is used to link the `Integer` object created at line 6 and the actual parameter of the call to `addElement`.

The program representation for the example is shown in Figure 3; for simplicity, the inverse edges are not shown. Each entry and exit edge is also treated as an assign edge for language L_{F} to represent parameter passing and method returns. The analysis can conclude that the points-to set of i_{12} has a single object o_6, because there exists a *flowsTo* path between them. To see this, first note that $this_{41}$ *alias* $this_{38}$ because object o_4 can flow to both $this_{41}$ and $this_{38}$. Similarly, $this_{44}$ *alias* $this_{38}$ and $this_{41}$ *alias* $this_{44}$ can be derived. Second, t_{41} and t_{44} are aliases because object o_{38} can flow to both t_{41} and t_{44}. For example, o_{38} *flowsTo* t_{41} can be derived as follows:

$$o_{38}\ \mathsf{new}\ t_{38}\ \mathsf{store(arr)}\ this_{38}\ alias\ this_{41}\ \mathsf{load(arr)}\ t_{41}$$
$$\Rightarrow\ o_{38}\ \mathit{flowsTo}\ t_{41}$$

Finally, o_6 *flowsTo* i_{12} can be derived as follows:

$$o_6\ \mathsf{new}\ tmp_6\ \mathsf{store(arr\_elm)}\ t_{41}\ alias\ t_{44}\ \mathsf{load(arr\_elm)}\ i_{12}$$
$$\Rightarrow\ o_6\ \mathit{flowsTo}\ i_{12}$$

In addition, this *flowsTo* path is a realizable interprocedural path because it contains matched pairs of entry and exit edges. The chain of entry and exit edges along the path is $\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30} \rightarrow \overline{\mathsf{entry}_{30}} \rightarrow \overline{\mathsf{entry}_6} \rightarrow \mathsf{entry}_4 \rightarrow \overline{\mathsf{entry}_4} \rightarrow \mathsf{entry}_8 \rightarrow \mathsf{entry}_{12} \rightarrow \mathsf{exit}_{12}$, which does not have any unmatched pair of method entry and return. (Recall that an $\overline{\mathsf{entry}}$ edge is treated as an exit edge.)

2.3 Formulation of Container Operations

This section presents our formulation of container operations based on the CFL-reachability formulation of points-to analysis.

DEFINITION 1. ***(Container)****. A* container type Γ *is an abstract data type with a set* Σ *of element objects, and two basic operations ADD and GET that operate on* Σ*. A* container object γ^n *is an instantiation of* Γ *with* n *elements forming an element set* Σ_γ*. ADD maps a pair of a container object and an element object to a container object. GET maps a container object to one of its elements. The effects of the operations are as follows:*

- $\mathit{ADD}(\gamma^n_{pre}, o) : \gamma^m_{post} \;\equiv\; o \notin \Sigma_{\gamma_{pre}} \wedge o \in \Sigma_{\gamma_{post}} \wedge m{=}n{+}1 \wedge \forall p : p \in \Sigma_{\gamma_{pre}} \rightarrow p \in \Sigma_{\gamma_{post}}$
- $\mathit{GET}(\gamma^n) : o \;\equiv\; o \in \Sigma_\gamma$

Following the common Java practice, the definition does not allow a container to have primitive-typed elements.[3] Furthermore, for the purposes of our tool, operations that reduce the size of a container are ignored. Despite their simplicity, these two abstract container operations capture many common usage scenarios for Java containers.

To identify semantics-achieving statements for a container, we first introduce relation *reachFrom* (short for "reachable from"). For each abstract object o, set $\{o' \mid o'\ \mathit{reachFrom}\ o\}$ consists of the object itself and other abstract objects whose run-time instances could potentially be reached from an instance of o through one or more level(s) of field deference(s). The semantic domains that will be used are defined in a standard way as follows:

$\mathsf{Obj}^\natural$:	Domain of abstract objects, as represented by object allocation sites
$\mathsf{C}^\natural \subset \mathsf{Obj}^\natural$:	Domain of container objects
$\mathsf{V}^\natural$:	Domain of variable identifiers
$\mathsf{F}^\natural$:	Domain of instance field identifiers
$\mathsf{Call}^\natural$:	Domain of method entry and exit edges

DEFINITION 2. ***(Relation*** *reachFrom). reachFrom* $\subseteq \mathsf{Obj}^\natural \times \mathsf{Obj}^\natural$ *is defined by the following production*

$$\mathit{reachFrom} \;\rightarrow\; \mathit{flowsTo}\ \mathsf{store(f)}\ \overline{\mathit{flowsTo}}\ \mathit{reachFrom} \mid \epsilon$$

In addition, the string consisting of entry *and* exit *edge labels on a reachFrom path has to be accepted by language* R_{C}*.*

For example, there exists a *reachFrom* path from o_6 to o_4. To see this, first note that o_{38} *reachFrom* o_4 holds, because of o_{38} *flowsTo* t_{38} store(arr) $this_{38}$ $\overline{\mathit{flowsTo}}$ o_4. Second, there exists

[3] For brevity, we may use "container" instead of "container object".

a *reachFrom* path from o_6 to o_{38}, because of o_6 *flowsTo* e_{40} $\mathsf{store(arr\_elm)}$ t_{41} $\overline{\textit{flowsTo}}$ o_{38}. Finally, o_6 *reachFrom* o_4 due to the transitive property of the relation. In addition, this entire *reachFrom* path does not contain any unmatched pair of method entry and exit.

Note that an object subgraph reachable from a container (i.e., containing only nodes relevant to the container) can be computed by searching for the *reachFrom* paths ending at the container object. Nodes irrelevant to the container can be filtered out by the context-sensitivity check of language R_C. Finding *reachFrom* paths is a single-target CFL-reachability problem with $O(n^3k^3)$ complexity, where n is the number of nodes in the flow graph and k is the size of language L_F. However, checking context sensitivity is exponential, as the size of language R_C is exponential in the size of the program (due to the exponential number of call chains). To ensure both high precision and scalability, a time constraint (discussed later) is imposed on the analysis to inspect each container in the program. If no valid *reachFrom* paths are found within the given time budget, the analysis gives up on the container and moves on to check the next one.

Based on *reachFrom*, we can distinguish *element objects* and *inside objects* from the set of all objects reachable from the container object.

DEFINITION 3. ***(Element object and inside object).*** *An object* $\mathtt{i} \in \mathsf{Obj}^\natural$ *is an inside object with respect to a container object* $\mathtt{c}$ *(where* $\mathtt{i}$ *is different from* $\mathtt{c}$*) if (1)* $(\mathtt{i}, \mathtt{c}) \in$ *reachFrom, and (2)* $\mathtt{i}$ *is created in the container class, its (direct or transitive) superclass, or any other class specified by the user.*

An object $\mathtt{e} \in \mathsf{Obj}^\natural$ *is an element object with respect to* $\mathtt{c}$ *if (1)* $(\mathtt{e}, \mathtt{c}) \in$ *reachFrom, (2)* $\mathtt{e}$ *is neither* $\mathtt{c}$*, nor an inside object of* $\mathtt{c}$*, and (3) for some reachFrom path from* $\mathtt{e}$ *to* $\mathtt{c}$*, all object nodes along the path (except for* $\mathtt{e}$ *and* $\mathtt{c}$*) are inside objects with respect to* $\mathtt{c}$*.*

In our example, o_4, o_{13} and o_{20} are container objects. Object o_6 is an element object for o_4; o_{14} and o_{15} are element objects for o_{13}; and o_{22} is an element object for o_{20}. Object o_{38} is an inside object for all three containers. In the rest of the discussion we will use I^c and E^c to denote the domains of inside objects and element objects with respect to a container $c \in \mathsf{C}^\natural$.

The definition of inside objects provides the flexibility to use a programmer-defined list that separates the client classes from the classes that are involved in the implementation of the container functionality. At present, the tool does not require such a list, as it targets only containers from the Java collections framework. In this case, it is sufficient to distinguish a client object from a Java collection internal object by checking if the object is created within the `java.util` package. (Note that the tool does not check the efficient use of containers within the JDK library code.) However, the class list will be useful if the tool is extended to inspect user-defined containers, because such containers may use an object created in a non-container class as an internal object. Such a (non-container) utility class should be explicitly listed as such.

The *semantics-achieving statements* are the loads/stores that read/write element objects from/to inside objects of a container.

DEFINITION 4. ***(Semantics-achieving statements).*** *A statement that achieves the functionality of ADD with respect to a container object* $\mathtt{c}$ *is a store of the form* $\mathtt{a.f} = \mathtt{b}$ *(where* $\mathtt{a}$*,* $\mathtt{b} \in \mathsf{V}^\natural$*,* $\mathtt{f} \in \mathsf{F}^\natural$*), such that there exists an addTo path from an object* $\mathtt{o_b}$ *that* $\mathtt{b}$ *points to, to the container object* $\mathtt{c}$*. This addTo path has the following components: (1) a flowsTo path between* $\mathtt{o_b}$ *and* $\mathtt{b}$*, (2) an edge* $\mathtt{b} \xrightarrow{\mathsf{store(f)}} \mathtt{a}$ *representing the store, (3) a* $\overline{\textit{flowsTo}}$ *path between* $\mathtt{a}$ *and an object* $\mathtt{o_a}$*, and (4) a reachFrom path from* $\mathtt{o_a}$ *to* $\mathtt{c}$*. Using* $\oplus$ *to denote path concatenation, the path is*

$$\begin{aligned} addTo(\mathtt{o_b}, \mathtt{c}) \triangleq\ & flowsTo(\mathtt{o_b}, \mathtt{b}) \oplus \mathtt{b} \xrightarrow{\mathsf{store(f)}} \mathtt{a} \oplus \overline{flowsTo}(\mathtt{a}, \mathtt{o_a}) \\ & \oplus reachFrom(\mathtt{o_a}, \mathtt{c}) \end{aligned}$$

where $\mathtt{o_a}$*,* $\mathtt{o_b} \in \mathsf{Obj}^\natural$*,* $\mathtt{o_a} \in \mathsf{I}^\mathsf{c}$*, and* $\mathtt{o_b} \in \mathsf{E}^\mathsf{c}$*.*

A statement that achieves the functionality of GET with respect to a container object $\mathtt{c}$ *is a load of the form* $\mathtt{b} = \mathtt{a.f}$ *(where* $\mathtt{a}$*,* $\mathtt{b} \in \mathsf{V}^\natural$*,* $\mathtt{f} \in \mathsf{F}^\natural$*), such that there exists a getFrom path from an object* $\mathtt{o_b}$ *that* $\mathtt{b}$ *points to, to the container object* $\mathtt{c}$ *where*

$$\begin{aligned} getFrom(\mathtt{o_b}, \mathtt{c}) \triangleq\ & flowsTo(\mathtt{o_b}, \mathtt{b}) \oplus \mathtt{b} \xrightarrow{\overline{\mathsf{load(f)}}} \mathtt{a} \oplus \overline{flowsTo}(\mathtt{a}, \mathtt{o_a}) \\ & \oplus reachFrom(\mathtt{o_a}, \mathtt{c}) \end{aligned}$$

where $\mathtt{o_a}$*,* $\mathtt{o_b} \in \mathsf{Obj}^\natural$*,* $\mathtt{o_a} \in \mathsf{I}^\mathsf{c}$*, and* $\mathtt{o_b} \in \mathsf{E}^\mathsf{c}$*.*

In addition, the string consisting of entry *and* exit *labels on an addTo or a getFrom path has to be accepted by language* R_C*.*

The goal of the analysis is to identify semantics-achieving statements by finding all *addTo* and *getFrom* paths for each container. An *addTo* path is a *reachFrom* path from o_b to c, which models the process of element object o_b being added to the container. Hence, the computation of *addTo* paths can be performed along with the computation of *reachFrom* paths. However, the computation of a *getFrom* path requires a priori knowledge of *reachFrom* paths and cannot be performed until all *reachFrom* paths, element objects, and inside objects are identified. In the running example, there exists an *addTo* path from o_6 to o_4, and the semantics-achieving statement on this path is the store $t[pos\text{++}] = e$. There also exists a *getFrom* path from o_6 to o_4, because of o_6 *flowsTo* *ret* and t_{44} $\overline{\textit{flowsTo}}$ o_{38} *reachFrom* o_4. The semantics-achieving statement on this path is the load *ret* = $t[index]$ at line 44.

The identification of semantics-achieving statements is not sufficient to understand the usage of a particular container object, as different container objects can have the same semantics-achieving statements. For example, all of the *addTo* (or *getFrom*) paths between o_6 and o_4, o_{14} and o_{13}, and o_{22} and o_{20} have $t[pos\text{++}]$ = e at line 41 (or *ret* = $t[index]$ at line 44) as their semantics-achieving statement. These statements are executed from multiple calling contexts and it is crucial to identify the contexts that correspond to the container object to be inspected.

DEFINITION 5. ***(Relevant context).*** *For each addTo or getFrom path* $\mathtt{p}$ *that ends at container object* $\mathtt{c}$*, let* $\mathtt{r}$ *be the prefix of* $\mathtt{p}$ *that appears before the semantics-achieving statement on* $\mathtt{p}$*. In other words,* $\mathtt{r}$ *is the flowsTo path before the corresponding store/load statement that achieves the ADD/GET functionality. Let* $\mathtt{l} \in \mathsf{Call}^{\natural\star}$ *be the chain of* entry *and* exit *edges (some of which may be inverted) along* $\mathtt{r}$*. The relevant context for the semantics-achieving statement on* $\mathtt{p}$ *is a sub-chain of* $\mathtt{l}$ *that contains only the unbalanced* entry *and* $\overline{\mathsf{exit}}$ *edges.*

The chain of entry and exit edges before the semantics-achieving statement on p represents the method invocations that cause the element object o_b to flow to variable b. This chain models the process of the element object being added to/retrieved from the container. The remaining entry and exit edges on p are irrelevant for this adding/retrieving process, because they represent calls that cause the inside objects (rather than the element objects) to flow into the container. We do not need to consider balanced entry/exit edges, as they represent completed invocations along the data flow. An example will be given shortly to illustrate this modeling. Note that there could be multiple relevant contexts for a semantics-achieving statement, because a container can have multiple element objects and each element objects can be added to (and retrieved from) the container through multiple calls.

Algorithm 1: Solving single source *addTo*-reachability.

```
    Input: Flow graph, container $c$, context-insensitive points-to solution $pts$
    Output: Map $solution$: pairs (heap store achieving ADD, relevant contexts)
1   Map⟨Statement, Set⟨Stack⟩⟩ $solution \leftarrow \emptyset$
2   Set⟨AllocNode⟩ $reachFrom \leftarrow \{c\}$        // reachable objects
3   Set⟨AllocNode⟩ $elemObj \leftarrow \emptyset$            // element objects
4   List⟨AllocNode⟩ $objectList \leftarrow \{c\}$              // worklist
5   List⟨Set⟨Stack⟩⟩ $contextSetList \leftarrow \{\{\text{EMPTY\_STACK}\}\}$
6   while $objectList \neq \emptyset$ do
7       remove an allocation node $o$ from the head of $objectList$
8       remove a set $contexts$ of context stacks from the head of
        $contextSetList$
9       Set⟨Stack⟩ $baseContexts \leftarrow \emptyset$
10      foreach store $a.f$ = $b$ such that $o \in pts(a)$ do
11          foreach context stack $s \in contexts$ do
12              $baseContexts \leftarrow baseContexts \cup$
                $\textsc{Compute}FlowsTo(o, a, s)$
13          Set⟨Stack⟩ $rhsContexts \leftarrow \emptyset$
14          foreach allocation node $o_b \in pts(b)$ do
15              foreach context stack $s \in baseContexts$ do
16                  $rhsContexts \leftarrow rhsContexts \cup$
                    $\textsc{Compute}\overline{FlowsTo}(b, o_b, s)$
17          if $rhsContexts \neq \emptyset$ then
18              $reachFrom \leftarrow reachFrom \cup \{o_b\}$
19              if ($o$ = $c$ OR $o$ is an inside object) AND ($o_b$ is NOT an inside
                object) then
20                  $elemObj \leftarrow elemObj \cup \{o_b\}$     // An element
                    object is found
21                  $solution \leftarrow solution \cup (a.f = b, rhsContexts)$
22              else
23                  $objectList \leftarrow append(objectList, o_b)$
24                  $contextSetList \leftarrow$
                    $append(contextSetList, rhsContexts)$
25  return $solution$
```

Algorithm 2: Solving single source *getFrom*-reachability.

```
    Input: Flow graph, container $c$, context-insensitive points-to solution $pts$,
           relation $reachFrom$, set $elemObj$
    Output: Map $solution$: pairs (heap load achieving GET, relevant contexts)
1   Map⟨Statement, Set⟨Stack⟩⟩ $solution \leftarrow \emptyset$
2   foreach allocation node $o \in elemObj$ do
3       foreach load $b = a.f$, such that $o \in pts(b)$ do
4           Set⟨Stack⟩ $lhsContexts \leftarrow \emptyset$
5           $lhsContexts \leftarrow \textsc{Compute}FlowsTo(o, b, \text{EMPTY\_STACK})$
6           Set⟨AllocNode⟩ $ins \leftarrow pts(a) \cap reachFrom$
7           Set⟨Stack⟩ $baseContexts \leftarrow \emptyset$
8           foreach AllocNode $o_a \in ins$ do // A candidate load
9               foreach context stack $s \in lhsContexts$ do
10                  $baseContexts \leftarrow baseContexts \cup$
                    $\textsc{Compute}\overline{FlowsTo}(a, o_a, s)$
11          if $baseContexts \neq \emptyset$ then
12              $solution \leftarrow solution \cup (b = a.f, lhsContexts)$
13  return $solution$
```

The chain of unbalanced entry and $\overline{\text{exit}}$ edges, together with the semantics-achieving statement, can be used to represent a specific (*ADD* or *GET*) operation executed on a specific container object. For example, the chain of entry and exit edges on the *addTo* path from o_6 to o_4 before semantics-achieving statement $t[pos$++$]$ = e is $\text{entry}_6 \longrightarrow \text{entry}_{30}$, which is the relevant context for the store operation with respect to container o_4. As another example, the chain on the *getFrom* path from o_6 to o_4 before the load operation ret = $t[index]$ is $\text{entry}_6 \longrightarrow \text{entry}_{30} \longrightarrow \overline{\text{entry}_{30}} \longrightarrow \overline{\text{entry}_6} \longrightarrow \text{entry}_4 \longrightarrow \overline{\text{entry}_4} \longrightarrow \text{entry}_8 \longrightarrow \text{entry}_{12}$. Hence, the relevant context for this *getFrom* path is $\text{entry}_8 \longrightarrow \text{entry}_{12}$, which captures the fact that element object o_6 is retrieved from container o_4. Later we will consider the relationship between the execution frequencies of statements $t[pos$++$]$ = e and ret = $t[index]$ only under their respective relevant contexts $\text{entry}_6 \longrightarrow \text{entry}_{30}$ and $\text{entry}_8 \longrightarrow \text{entry}_{12}$.

2.4 Analysis Algorithms

The algorithms for solving *addTo*- and *getFrom*-reachability are shown in Algorithm 1 and Algorithm 2, respectively.

Both algorithms rely on an initial context-insensitive points-to set to find candidates for semantics-achieving statements. Algorithm 1 iteratively computes a set of reachable objects. The i-th element of list *contextSetList* keeps a set of relevant contexts for the i-th object in worklist *objectList*. Each context is represented by a stack, which contains exactly the chain of unbalanced entry and $\overline{\text{exit}}$ edges of a *flowsTo* path. Initially, *objectList* contains the container object c and *contextSetList* contains an empty stack. Map *solution* contains pairs of semantics-achieving statement and relevant contexts, which will be returned after the function finishes.

Function $\textsc{Compute}FlowsTo$ (o, a, s) at line 12 attempts to find *flowsTo* paths from an object o to a variable a, under calling context s that leads to the method creating o. Due to space limitations this function is not shown; conceptually, it is similar to the FIND-POINTSTO algorithm described in [37]. The function returns a set of contexts (i.e., stacks) that are chains of unbalanced edges on the identified *flowsTo* paths from o to a. An empty set returned means that there does not exist any valid $\overline{flowsTo}$ path between them.

Similarly, function $\textsc{Compute}\overline{FlowsTo}$ (b, o_b, s) at line 16 attempts to find a $\overline{flowsTo}$ path from variable b to object o_b, under calling context s that leads to the method declaring b. The purpose of this function is to connect the chain of entry and exit edges on the *flowsTo* path from o to a with the chain of entry and exit edges on the $\overline{flowsTo}$ path from b to o_b, and to check if the combined chain corresponds to a realizable call path. If the combined chain is a realizable path (i.e., $rhsContexts \neq \emptyset$ at line 17), o_b is added to set *reachFrom* of reachable objects. Furthermore, if o_b is found to be an element object (line 20 and line 21), it is included in set *elemObj*, which will be used later by Algorithm 2. At this time, it is clear that store $a.f$ = b is a semantics-achieving statement, and its relevant contexts are contained in set *rhsContexts*. If o_b is not an element object, we append o_b to the worklist (and also append *rhsContexts* to *contextSetList*) for further processing. Some subsequent iteration of the *while* loop will use this context set to compute *flowsTo* path from object o_d to variable d for a new store $c.f$ = d (line 12), etc. In this case, remembering and eventually using *rhsContexts* ensures that no unrealizable paths can be produced during the discovery of the container's data structure.

Note that we omit a check for recursive data structures in the algorithm. In fact, an object is not added into the worklist, if it has been visited earlier during the processing of reachable objects. In other words, the back reference edges between inside objects in the object graph are ignored, because they have nothing to do with the element objects. It is also possible for an element object to have a back reference edge going to an inside object or the container object (although this is not likely to happen in practice). This back edge is also ignored, because we are interested in the process where the element object is added to the container, rather than in the shape of the data structure of the container object.

Algorithm 2 inspects each element object o computed in Algorithm 1 (line 2) and attempts to find load statements of the form b = $a.f$ such that b could point to o. As before, the algorithm starts from a context-insensitive solution, and then checks if there exists a *flowsTo* path from o to b (line 5). If such *flowsTo* paths are found, it uses the set of contexts returned (i.e., chains of unbalanced edges extracted from these paths) to compute $\overline{flowsTo}$ paths from a to object o_a that a may potentially point to. Note that we use an intersection between $pts(a)$ and $reachFrom$ (line 6) to filter out irrelevant objects that are not reachable from container c. If a $\overline{flowsTo}$ path can be found (line 11-line 12), this load is a semantics-achieving statement and the relevant contexts for it are contained in the set $lhsContexts$.

Precision improvement Not all load statements identified in *getFrom* paths correspond to *GET* operations. For example, methods `equals` and `remove` in many container classes need to load element objects for comparison (rather than returning them to the client). To avoid the imprecise results generated in these situations, we employ a heuristic when selecting statements that implement *GET* operations. A load $b = a.f$ is selected if (1) it is on a valid *getFrom* path, and (2) the points-to set of b and the points-to set of the return variable of the method where the load is located have a non-empty intersection. This heuristic is based on the common usage of Java containers: only methods that can return element objects can be used to retrieve objects by a client.

It would be interesting to investigate other heuristics in future refinements of the analysis. In situations where individual statements are not precise enough to capture the semantics of *ADD* and *GET*, it may be possible to find courser-grained program entities (e.g., methods) that correspond to these abstract operations. For example, a splay tree may perform both stores and loads for a single *GET* operation. At present, it is unclear how to generalize the analysis to model such cases.

3. Execution Frequency Comparison

This section describes the dynamic profiling algorithm and the static inference algorithm to compare the execution frequencies of semantics-achieving statements.

3.1 Dynamic Frequency Profiling

During the execution, the profiling framework needs to record, for each container object, its *ADD* and *GET* frequencies. A key challenge is how to instrument the program in a way so that the frequencies of semantics-achieving statements can be associated with their corresponding container objects. In many cases, the container object is not visible in the method containing its semantics-achieving statements. For example, the store that implements *ADD* for `HashMap` is located in the constructor of class `HashMap.Entry` where the root `HashMap` object cannot be referenced. In this and similar cases, it is unclear where the instrumentation code should be placed to access the container object.

We use relevant contexts to determine the instrumentation points. Given a pair $(s, e_0, e_1, \ldots, e_n)$ of a semantics-achieving statement s and its context, we check whether the receiver of each call site e_i can be the container object c. This check is performed in a bottom-up manner (i.e., from n down to 0). The instrumentation code is inserted before the first call site $e_i : a.f()$ found during the check such that the points-to set of a includes c. For example, one instrumentation site for `HashMap` is placed before the call to `addEntry` in method `put`, because the receiver variable of the call site can point to the `HashMap` object:

```
class HashMap{ ...
   void put(K key, V value){ ...
      // increment the ADD frequency for "this" container
      recordADD(this);
      this.addEntry(..., key, value, ...);
   }
   void addEntry(..., K key, V value, ...){ ...
     table[...] = new Map.Entry(..., key, value, ...);
   }
}
class Entry{
   Entry(..., K key, V value, ...){
      // these are stores achieving ADD
      this.key = key;
      this.value = value;
   }
}
```

3.2 Static Inference of Potentially-Smaller Relationships

This subsection describes the static inference algorithm that detects inefficiencies by inferring potentially-smaller/larger relationships for the execution frequencies of two (semantics-achieving statement, contexts) pairs. These relationships are computed by traversing an interprocedural *inequality graph*, which models the interprocedural nesting among the loops containing the semantics-achieving statements.

DEFINITION 6. ***(Inequality graph).*** *An inequality graph* $IG = (\mathcal{N}, \mathcal{E})$ *has node set* $\mathcal{N} \subseteq \mathcal{L} \cup \mathcal{M}$*, where* $\mathcal{L}$ *is the domain of loop head nodes, and* $\mathcal{M}$ *is the domain of method entry nodes. The edge set is* $\mathcal{E} \subseteq \mathcal{C} \cup \mathcal{I}$*, where* $\mathcal{C}$ *represents call edges and* $\mathcal{I}$ *represents inequality edges.*

3.2.1 Inequality Graph Construction

For two statements s_1 and s_2 that are located in loops l_1 and l_2 respectively, we say that the execution frequency of s_1 is *potentially-smaller* than the execution frequency of s_2 if l_2 is nested within l_1. Such a relationship does not necessarily reflect the real run-time execution frequencies (thus the use of "potentially"). For example, if s_2 is guarded by a non-loop predicate inside l_2, it is possible that it is executed less frequently than s_1 because the path containing s_2 can be skipped many times inside the body of l_2. One example of this situation comes from a common container implementation scenario. When the client attempts to add an object into the container, the implementation first checks if the object is already in the container, and stores it only if the container does not have it already. In this situation, the semantics-achieving statement is under the non-loop predicate that checks whether the element object is already in the container.

Despite this potential imprecision, this modeling of execution frequency, to a large degree, captures the high-level programmer's intent. For example, in many cases the programmer just wants to add objects using the nested loops without even caring about whether they have been added before. Even though the statically-inferred potentially-smaller relationship may not hold for some particular runs of a program, the problems found using this relationship may reflect inefficient uses of containers in general. In addition, the loop nesting relationship itself may clearly suggest a fix if a problem really exists. For example, the problem may be solved simply by pulling some operations out of a loop. We have found this approach based on loop nesting to work well in practice.

The algorithm for constructing the inequality graph is shown in Algorithm 3. For each method m in the call graph, intraprocedural inequality edges are first added (line 4-line 11). For each loop head, we find the loop in which it is nested (line 6). If it is not nested in any loop (line 7-line 8), we create an inequality edge between the entry node of the method and the loop head node. Otherwise, the edge is created between the head of the surrounding loop and the node (line 9-line 11).

For each caller of m, a call edge is added to connect the two methods (line 14-25). Specifically, the loop where the call site for m is located (or the entry node of the caller) is found (line 17), and a call edge is created to link the head of the loop (or the entry node of the caller) and m's entry node. Call edges are useful in filtering out irrelevant calling contexts during the traversal of the inequality graph. Figure 4 shows an inequality graph for the running example.

Algorithm 3: Algorithm for constructing the inequality graph.

```
Input: Call graph $CG$
Output: Inequality graph $IG$
foreach method $m$ in the call graph do
    EntryNode $entry \leftarrow$ GETENTRYNODE($m$)
    CFG $cfg \leftarrow$ BUILDCFG($m$)
    foreach loop $l \in cfg$ do                         // add inequality edges
        LoopHeadNode $head \leftarrow$ GETLOOPHEADNODE($l$)
        Loop $l' \leftarrow$ FINDSURROUNDINGLOOP($head$, $cfg$)
        if $l'$ = null then
            CREATEINEQUALITYEDGE($entry \xrightarrow{\leq} head$)
        else
            LoopHeadNode $head' \leftarrow$ GETLOOPHEADNODE($l'$)
            CREATEINEQUALITYEDGE($head' \xrightarrow{\leq} head$ )
    foreach incoming call graph edge $e$ do             // add call edges
        Method $caller \leftarrow$ SOURCE($e$)
        CFG $cfg \leftarrow$ GETCFG($caller$)
        Loop $l$ = FINDSURROUNDINGLOOP($e.callsite$, $cfg$)
        if $l$ = null then
            EntryNode $entry' \leftarrow$ GETENTRYNODE($caller$)
            CREATECALLEDGE($entry' \xrightarrow{call(e)} entry$)
        else
            LoopHeadNode $head \leftarrow$ GETLOOPHEADNODE($l$)
            CREATECALLEDGE($head \xrightarrow{call(e)} entry$)
```

Here we use e_i to denote the entry node for the method declared at line i, and l_i to denote the loop head node located at line i. Each call edge is annotated with call_i, which represents the call site at line i. Each inequality edge is annotated with $\leq_i$, where i is a globally-named index. Inverse edges are allowed for call edges: if there is a call edge call_e between nodes m and n, an edge $\overline{\mathsf{call}_e}$ exists between n and m. Unlike in the flow graph, there are no exit edges in the inequality graph because it is not necessary to model any data flow.

An inequality edge is used to represent only potentially-smaller relationships. The potentially-larger relationships could potentially be represented by inverse edges; however, we do not allow the use of such inverse edges because a path in the graph must represent only one of these two relationships (i.e., either smaller or larger, but not both).

DEFINITION 7. ***(Valid potentially-smaller path).*** *Given two inequality graph nodes* `m` *and* `n`*, a path* `p` *from* `m` *to* `n` *is a valid potentially-smaller path if the chain of call edges (including inverse edges) on* `p` *forms a realizable interprocedural path (i.e., the sequence of edge labels on the chain forms a string in language* R_{C}*). Path* `p` *is a strictly-smaller path if (1)* `p` *is a valid potentially-smaller path and (2)* `p` *contains at least one inequality edge.*

One can easily define a grammar with starting non-terminal *potentially-smaller* to capture the above definition of validity. Similarly to the *flowsTo* computation, finding a valid *potentially-smaller* path in the inequality graph requires a context-sensitivity check of call and $\overline{\mathsf{call}}$ edges.

3.2.2 Inefficiency Detection as Source-Sink Problems

Detecting underutilized containers To find an underutilized container, we need to compare the execution frequencies of the container allocation site c and each store s that implements the functionality of *ADD* under relevant context r with respect to c. In the following definition, lh(`s`) denotes the loop head (if `s` is within a loop) or the entry node of the method (if `s` is not in a loop) for statement `s`.

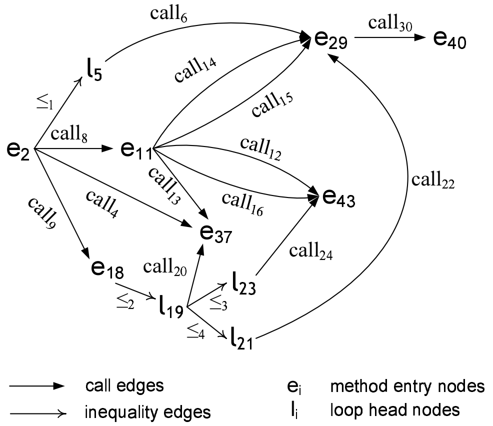

Figure 4. Inequality graph for the running example.

DEFINITION 8. ***(Underutilized container).*** *Given a container allocation site* `c` *and a set of statement-contexts pairs that implement ADD operations for* `c`*, an underutilized container problem occurs for* `c` *if there does* ***not*** *exist a pair* (`store`, `contexts`) *for which (1) a strictly-smaller path* `p` *exists from* lh(`c`) *to* lh(`store`)*, and (2) there exists a context* `t` $\in$ `contexts` *such that* `p` *ends with the chain of call edges represented by* `t`.

Informally, an underutilized container problem is reported if there does not exist an *ADD* operation such that the loop where it is located is nested within the loop where the container allocation site is located.

Consider again the running example. Recall that the statement-contexts pair that achieves *ADD* operation for container o_4 is ($t[pos{+}{+}] = e$, {$\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}$}). There exists a strictly-smaller path $\leq_1\rightarrow$ call_6 $\rightarrow$ call_{30} from node e_2 to e_{40}, which are the method entry nodes for the allocation site of o_4 (line 4) and for $t[pos{+}{+}] = e$ (line 41), respectively. In addition, this path contains the call chain $\mathsf{call}_6 \rightarrow \mathsf{call}_{30}$, which is exactly the context contained in set {$\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}$} in the pair (comparing only labels on the call edges and the entry edges). Hence, no underutilized container problem will be reported for container o_4.

As another example, the statement-contexts pair for *ADD* on o_{13} is ($t[pos{+}{+}] = e$, {$\mathsf{entry}_{14} \rightarrow \mathsf{entry}_{30}$, $\mathsf{entry}_{15} \rightarrow \mathsf{entry}_{30}$}). The tool reports that o_{13} exhibits an underutilized container problem, because no strictly-smaller paths from the entry node e_{11} of its allocation site (line 13) to the entry node e_{40} of $t[pos{+}{+}] = e$ can be found, under relevant calling context $\mathsf{entry}_{14} \rightarrow \mathsf{entry}_{30}$ or $\mathsf{entry}_{15} \rightarrow \mathsf{entry}_{30}$. The problem does not occur for container o_{20}, since there is a strictly-smaller path $\leq_4\rightarrow$ call_{22} $\rightarrow$ call_{30} from the allocation site at line 20 to the store operation $t[pos{+}{+}] = e$ that implements *ADD*, under the relevant calling context $\mathsf{entry}_{22} \rightarrow \mathsf{entry}_{30}$.

Note that although the number of elements added to container o_{13} (i.e., 2) is in fact larger than the number of times its allocation site can be executed (i.e.,1), it is not a false positive to report it as an underutilized container. This is because the creation of the container (which could cost hundreds of run-time instructions) can be easily avoided by introducing extra variables for storing the data. It could be the case that, while the *ADD* operations for a container are in the same loop as the allocation site of the container, it may not be easy to perform an optimization because there could be a large number of distinct *ADD* operations (e.g., the programmer intends to add many elements without using loops). However, we have found that this situation rarely occurs in real-world programs. Once an underutilized container problem is reported, there is usually an obvious container that holds a very small number of elements, and a specialization can be easily created.

Detecting overpopulated containers To find an overpopulated container, it is necessary to compare the number of *GET* operations against the number of *ADD* operations for the container.

DEFINITION 9. ***(Overpopulated container).*** *Given a container allocation site* $\mathtt{c}$*, a set* $\mathtt{S}_1$ *of statement-contexts pairs that implement ADD for* $\mathtt{c}$*, and a set* $\mathtt{S}_2$ *of statement-contexts pairs that implement GET for* $\mathtt{c}$*,* $\mathtt{c}$ *is an overpopulated container if for any pair* $(\mathtt{store}, \mathtt{contexts}_1) \in \mathtt{S}_1$ *and any pair* $(\mathtt{load}, \mathtt{contexts}_2) \in \mathtt{S}_2$*, (1) there exists a valid potentially-smaller path* $\mathtt{p}$ *from* $lh(load)$ *to* $lh(store)$*, and (2) there exist a context* $\mathtt{t}_1 \in \mathtt{contexts}_1$ *and a context* $\mathtt{t}_2 \in \mathtt{contexts}_2$ *such that* $\mathtt{p}$ *starts with the chain of (inverse) call edges represented by* $\overline{\mathtt{t}_2}$ *and ends with the chain of call edges represented by* $\mathtt{t}_1$.

Informally, an overpopulated container is reported if for every pair of *GET* and *ADD* operations, a potentially-smaller relationship can be inferred between them.

In the running example, an overpopulated container problem is detected for container o_4. Recall that the statement-contexts pair that implements *ADD* is ($t[pos$++$] = e$, $\{\mathsf{entry}_6 \rightarrow \mathsf{entry}_{30}\}$), and the statement-contexts pair that implements *GET* is ($ret = t[index]$, $\{\mathsf{entry}_8 \rightarrow \mathsf{entry}_{12}\}$). There exists an *potentially-smaller* path between these two statements: $\overline{\mathsf{call}_{12}} \rightarrow \overline{\mathsf{call}_8} \rightarrow\leq_1\rightarrow \mathsf{call}_6 \rightarrow \mathsf{call}_{30}$. This path contains the call edges $\mathsf{call}_6 \rightarrow \mathsf{call}_{30}$ (i.e. t_1) and $\overline{\mathsf{call}_8} \rightarrow \overline{\mathsf{call}_{12}}$ (i.e., $\overline{t_2}$). Container o_{13} is also overpopulated, because there exists a valid *potentially-smaller* path from the *GET* to any of the two *ADD* operations.

Container o_{20} is not overpopulated, since no *potentially-smaller* path can be found from its *GET* operation (i.e., the statement-contexts pair ($ret = t[index]$, $\{\mathsf{entry}_{24}\}$)) to its *ADD* operation (i.e., pair ($t[pos$++$] = e$, $\{\mathsf{entry}_{22} \rightarrow \mathsf{entry}_{30}\}$)).

3.2.3 Analysis Algorithm

In general, both the proof and the disproof of a certain path under certain calling contexts requires a traversal of the inequality graph. The traversal has to follow the call edges represented by a start context and an end context, which are the relevant contexts associated with semantics-achieving statements. The start context is an empty stack if the statement is the allocation site of the container. A standard worklist-based algorithm is used in the expected way to perform a breadth-first traversal of the graph. The traversal terminates immediately if call edges in the path are detected to form a cycle, because there is no way to reason about the number of *ADD* and *GET* operations for a container if calls connecting these operations are involved in recursion.

Trade-off between the analysis scalability and the amount of information produced Because the inequality graph does not contain any data flow information, the graph traversal algorithm can follow arbitrary call and $\overline{\mathsf{call}}$ edges when selecting the path. The start and end contexts are useful when the algorithm attempts to decide which call/$\overline{\mathsf{call}}$ edge to follow. Suppose the algorithm is inspecting method m. The algorithm decides to leave m through a call edge if (1) this edge is on the end context, or (2) it can lead to the end context. On the other hand, it follows a $\overline{\mathsf{call}}$ edge if (1) the edge is on the start context, or (2) there does not exist any call edge going out of m that is on the end context or can lead to the end context. A call edge is "leading to" a context when the method that the call edge goes to can (directly or transitively) invoke the source method of the first call edge on the context. In addition, we keep track of the set of methods that the related *addTo* and *getFrom* paths have passed. The traversal of the inequality graph never enters a method if this method is not in this set.

While the start and end contexts are useful, the worst-case time complexity of a naive graph traversal algorithm is still exponential, as the number of distinct calling contexts is exponential in the size of the program. This motivates the need to define a trade-off framework to handle the analysis scalability and the amount of information produced.

One factor considered by this framework is the number of unbalanced $\overline{\mathsf{call}}$/call edges in a valid *potentially-smaller* path. These numbers represent the length of the method sequences on the call stack that the path crosses, and they implicitly determine the running time of the algorithm. A path crossing too many calls is usually an indicator of an unrealizable interprocedural path (e.g., due to spurious call graph edges). Furthermore, even though an inefficiently-used container can be found by traversing a long interprocedural path on the inequality graph, it may be hard to optimize it as the data it carries might be needed by many places in the program. In this framework, the number of unbalanced $\overline{\mathsf{call}}$/call edges allowed in a path can be pre-set as a threshold. While this introduces unsoundness, it improves the scalability and presents to the user a set of containers that are potentially easy to specialize.

Another factor (orthogonal to the number of unbalanced $\overline{\mathsf{call}}$/call edges) that is taken into account is the time used to inspect each container. If a `TimeOutException` is caught during the inspection of a container, the analysis moves on to the next container, without generating any warnings about this current one. We experimented with different time thresholds, and some of these results are described in Section 4.

3.3 Comparison between Static Inference and Dynamic Profiling

Compared with all existing bloat detection techniques based on dynamic analysis [22, 23, 26, 35, 42], a major weakness of a static analysis approach is its inability to estimate precisely the execution frequencies of various statements. However, it has the following three advantages over the dynamic approaches. First, the static analysis can be used as a coding assistance tool to find container-related problems during development, before testing and tuning have begun. It is desirable to avoid inefficient operations early, even before meaningful run-time executions are possible. Second, a problem detected by the static analysis usually indicates a programmer intent (or mistake) that is inherent in the program, while the results from a dynamic analysis depend heavily on the specific run-time execution being observed. Finally, the process of locating the underlying cause from the dynamically-observed symptoms is either completely manual or involves ad hoc techniques that do not quickly lead a tool user to the problematic code. For example, a profiler can find a container exhibiting few lookup operations, but it is hard for it to effectively explain this behavior to the programmer. The static tool explicitly reports the loops that cause the generation of the warnings, thus reducing the effort to "connect the dots" from the manifestation of the problem to the core cause.

4. Empirical Evaluation

We have implemented the static and dynamic analyses based on the Soot program analysis framework [41], and evaluated their effectiveness on the set of 21 Java programs shown in Table 1. All experiments used a dual-core machine with an Intel Xeon 2.80GHz processor, running Linux 2.6.9 and Sun JDK 1.5.0 with 4GB of max heap space. The Sridharan-Bodik analysis framework from [37] was adapted to compute CFL-reachability. A parallel version of the analysis was used: 4 threads were ran to simultaneously inspect containers. Note that the total numbers of reachable methods for some programs are significantly larger than the numbers shown in previous work [37] for the same programs. This is because of the

Benchmark	*#M(K)*	*#Con*	T_1 = 20 min					T_2 = 40 min					*Dynamic vs Static*				
			#UC	*#OC*	*#NC*	*#FP*	*RT(s)*	*#UC*	*#OC*	*#NC*	*#FP*	*RT(s)*	*#DU*	*#DO*	*#SN*	*#DN*	*#MS*
jack	12.5	34	8	4	2	*	1725	8	4	0	*	2561	8	4	*	*	*
javac	13.4	45	12	10	5	*	5040	12	10	5	*	5040	12	10	*	*	*
soot-c	10.4	15	0	1	3	0	1235	0	1	0	0	1440	0	1	0	0	0
sablecc-j	21.4	29	3	5	0	1	1140	-	-	-	-	-	3	5	0	4	0
jess	12.8	4	2	2	0	0	790	-	-	-	-	-	2	2	0	0	0
muffin	21.4	108	4	7	78	0	28213	5	16	56	0	66472	5	16	0	18	2
jb	8.2	9	1	7	0	0	64	-	-	-	-	-	1	7	0	0	0
polyglot	8.6	18	1	6	1	1	1259	0	6	1	0	2447	0	6	2	1	1
jflex	20.2	44	2	5	17	0	6785	2	5	14	0	16092	2	5	1	8	0
jlex	8.2	16	1	0	0	0	474	-	-	-	-	-	1	0	0	0	0
java-cup	8.4	10	1	3	0	0	519	-	-	-	-	-	1	3	0	0	0
antlr	12.9	15	2	2	0	0	584	-	-	-	-	-	2	2	0	2	0
bloat	10.8	260	18	46	118	$1^\sharp$	34542	24	76	17	$2^\sharp$	49844	24	72	2	18	1
chart	17.4	286	15	29	52	$1^\sharp$	26406	21	38	12	$1^\sharp$	35406	21	36	3	16	1
xalan	12.8	1	0	1	0	0	222	-	-	-	-	-	0	1	0	0	0
hsqldb	12.5	1	0	0	0	0	99	-	-	-	-	-	0	0	0	0	0
luindex	10.7	1	0	1	0	0	69	-	-	-	-	-	0	1	0	0	0
ps	13.5	42	0	8	0	0	1077	-	-	-	-	-	0	8	0	5	0
pmd	15.3	39	8	7	0	0	1322	-	-	-	-	-	5	7	0	19	0
jython	27.5	75	5	26	5	$0^\sharp$	7055	5	26	1	$0^\sharp$	9745	5	21	1	17	2
eclipse^1	41.0	1623	18	25	1097	$0^\sharp$	447465	18	32	956	$0^\sharp$	825897	18	27	0	20	0
eclipse^2	41.0	1623	47	121	351	$3^\sharp$	32151	47	137	104	$4^\sharp$	57897	45	110	5	20	0

Table 1. *#M* is the number of methods (in thousands) in Soot's Spark context-insensitive call graph. *#Con* is the total number of containers inspected in the application code. There are two rows for `eclipse`: (1) analyzing all plugins together and (2) analyzing them one at a time. Results are shown with T_1 = 20 minutes and T_2 = 40 minutes limit for the static tool to inspect each container. The last part of the table compares the static and dynamic analyses.

use of the JDK 1.5.0 library which is much larger than the JDK 1.3 library used in that previous work. Many of the programs were chosen from the DaCapo [2] benchmark set. The analysis was able to run on all of the DaCapo programs, but we excluded from the table the programs that do not use any Java containers. For `eclipse`, we analyzed the main framework and the following plugins that are necessary for the DaCapo run: org.eclipse.jdt, org.eclipse.core, org.eclipse.text, org.eclipse.osgi, and org.eclipse.debug.

Static analysis The first part of Table 1 (T_1 and T_2) shows the warnings generated by the static tool, the false positives, and the running times for two different configurations: 20 and 40 minutes allowed to inspect each container. If all containers can be completely inspected under the first configuration, the second configuration is not applied, and "-" marks the corresponding column. The table shows the number of underutilized container warnings (*#UC*), the number of overpopulated container warnings (*#OC*), the number of containers whose inspection is not completed due to time out (*#NC*), the number of false positives (*#FP*), and the total running time in seconds (*RT*). For programs with many warnings, we randomly picked 20 warnings (including both types of problems) for manual checking; the numbers of false positives found in these samples are reported and marked with $^\sharp$. The analysis running time shown in the table includes the identification of semantics-achieving statements and the inference of potentially-smaller relationships. They are not listed separately because the running time is dominated by the former.

We have tuned the analysis by adjusting the maximum number of unbalanced $\overline{\text{call}}$/call edges traversed. All numbers from 3 to 10 were evaluated. We observed that problems caused by certain container usage patterns are missing in the reports when this parameter is set to a number less than 7. Hence, 7 was chosen as the parameter value for the experiments. This value appears to be an appropriate choice for the set of benchmarks we used, and it may need to be re-adjusted for programs with different container usage patterns.

False positives are determined by manually inspecting each program. Given a warning, we examine the code and check whether the container is appropriately used, and whether there is a better way of using it. This is a subjective choice in which we are trying to simulate what an "intelligent programmer" would do. Although there is no objective perfect answer, such an evaluation provides valuable indication of the real-world usefulness of the tool. For programs such as `jack` and `javac` whose source code is not publicly available, we mark the *#FP* columns with "*", meaning that the warnings are not checked. Overall, the results of study are promising, since the number of false positives is low across the benchmark set.

In general, false negatives can be introduced by the unsoundness of the analysis. For example, a container problem can be missed if the container structure cannot be completely discovered within the allowed time limit, or if the call chain required to expose the problem is too long. The numbers of false negatives were not investigated because they can be reduced by increasing the resource budget (i.e., time limit and maximum call chain length). For example, for most programs in our study, a budget of T = 40 min allows the analysis to successfully inspect all containers in the application code (i.e., to achieve *#NC* = 0).

The tool could not finish the inspection of many container objects in `eclipse`^{1}. This performance is caused in part by the extremely large code base of `eclipse`, and in part by the lack of precise contexts in many *flowsTo* or $\overline{flowsTo}$ paths computed by the underlying Sridharan-Bodik framework (because of the use of "match edges" [37]). In our current work we decided to use the framework as-is and to focus on demonstrating that the approach successfully identifies problematic containers. Future work can improve the Sridharan-Bodik machinery to provide more precise context information for our analysis.

Splitting a large code base for scalability Note that both analysis time and the number of timed-out containers decrease substantially when `eclipse` plugins are analyzed individually (row `eclipse`^{2}). This is because the number of calling contexts for each container method is reduced significantly. In general, it is *not* always valid to analyze separately the components of a large program. However, separate analysis can be made more general by first employing a relatively inexpensive escape analysis, which identifies container objects that may be passed across the boundaries of components (or plugins). A container that can never escape the component where it is created can be safely analyzed in the absence of other components. This is similar to the observation that context sensitivity is not necessary for an object that never escapes its creating method. While element objects could still flow in and out of components, there exist techniques to handle incomplete programs, for example, by creating placeholders for missing objects [33]. For a non-escaping container, all semantics-achieving statements under the relevant calling contexts triggered by the creating component would be confined to that component.

Comparison with a dynamic approach The dynamic analysis instruments each semantics-achieving statement, runs the program, and reports containers whose (1) *ADD* frequencies are smaller than 10, and (2) *ADD* frequency/*GET* frequency ratios are greater than 2. The intersections between the sets of statically and dynamically generated warnings are shown in the last part of Table 1: *#DU* and *#DO* are the numbers of containers reported by the static analysis (from *#UC* and *#OC*, respectively) that also appear in the dynamic analysis reports. Note that most inefficiently-used containers found by the static analysis are also reported by the dynamic analysis, which shows that the static warnings indeed produce containers that exhibit problematic run-time behavior.

It is not as easy to use the dynamic analysis to find containers that are optimizable across all inputs and runs, compared to using the static analysis. Columns *#SN* and *#DN* show the numbers of containers reported by the static analysis and the dynamic analysis, respectively, for which we did *not* manage to come up with optimization solutions. To determine *#SN*, we examined each container that was already subjected to a manual check for false positives (with $T = 40$ min, and with 20 randomly chosen containers for programs with more than 20 warnings). To determine *#DN*, we examined all dynamically-reported containers, if there were at most 20 of them; otherwise, we examined the 20 containers with the highest potential for performance improvement: the ten containers with the largest number of *ADD* operations, and the ten containers with the largest ratio of *ADD* to *GET* operations. Among all these examined containers, *#DN* is the number of those for which we could not determine an appropriate optimization.

In the course of this experiment, it became clear that the problems reported by the static analysis are easier to fix than those reported by the dynamic analysis. For instance, among the top 20 dynamically-reported containers for `bloat`, a program analysis framework in DaCapo, we eventually came up with optimization solutions for only two, and one of them was also in the static analysis report. The remaining containers are used to hold various kinds of program structures such as CFGs and ASTs. While they are not retrieved frequently in one particular run (with the inputs provided by DaCapo), it is hard to optimize them as their elements may be heavily used when the program is run with other inputs. In contrast, most of the container problems reported by the static analysis are straightforward and the programmer can quickly come up with optimization solutions after she understands the loop nesting relationships that cause the tool to report the warnings.

While static analysis reports can precisely pinpoint the causes of inefficiency problems, they cannot say anything about the severity of the warnings. When the analysis is used during development (i.e., without any test runs) to find container-related problems, this information may not be required because a programmer should, ideally, fix all reported warnings to avoid potential bloat. However, ranking of warnings becomes highly necessary when the tool is used for performance tuning and problem diagnosis, as the reported warnings have different performance impact and importance. For example, when we attempt to find optimization opportunities for `bloat`, it is unclear, among the total of 100 warnings generated by the static analysis, which ones to inspect first. It is impractical to examine and fix all of them, a task that can be very labor-intensive and time-consuming. A natural solution is to rank the statically-reported containers based on their run-time allocation frequencies; this was the approach used in the two case studies described below.

As discussed earlier, the time budget limitations and the constraints on call chain length may cause the static analysis to miss problematic containers that are (mis-)used in complex ways. Consider the "top" inefficiently-used containers from the dynamic reports (the same 20 or less containers examined when determining *#DN*). Column *#MS* shows the number of such containers that are missed by the static analysis. While there are only few missed containers, a more comprehensive study of their properties could be performed in future work, in order to identify new usage patterns that may be used to refine our current static analysis.

Performance improvement from specializing containers For `bloat` we found that among the containers that have warnings, the most frequently-allocated container is an `ArrayList` created by method `children` for each expression tree node to provide access to its children nodes. The number of objects added in this container is always less than three, and in many cases, it does not contain any objects. We studied the code and found an even worse problem: even when the children do not change, this container is never cached in the node. Every time method `children` is invoked, a new list is created and the node's children are added. By creating a specialized version that takes advantage of container types such as `Collections.emptyList` and `Collections.singletonList` and that caches the children list in each node, we were able to reduce the number of objects created from 129253586 to 89913518 (30% reduction), and the running time from 147 seconds to 110 seconds (24.5% reduction).

We have also inspected the report generated for `chart`. Many problems reported are centered around method `getChunks` declared in an interface, which returns an `ArrayList`. This interface is implemented by more than 10 classes, each of which has its own implementation of `getChunks`. While many of these concrete classes do not have any chunks associated, they have to create and return an empty `ArrayList`, in order to be consistent with the interface declaration. By further studying the clients that invoke these `getChunks` methods, we found that many of them need only to know the number of chunks (i.e., by invoking `getChunks().size()`). We quickly modified the code to replace the empty `ArrayList` with the specialized `Collections.emptyList`, add a method `getSize` in each of the corresponding classes that calculates the number of chunks without creating a new list, and replace the calls `getChunks().size()` with calls to the new `getSize` method. This process took us less than an hour. The modified version achieved 3.5% running time reduction and 5% reduction of the number of generated objects. Note that these case studies only addressed two obvious problems and did not go into any depth; in general, significant optimizations may be possible if a developer familiar with the code thoroughly analyzes the reported inefficiently-used containers and creates appropriate specialized versions.

5. Related Work

Bloat detection Mitchell *et al.* [24] propose a manual approach that detects bloat by structuring behavior according to the flow of information, and their later work [23] introduces a way to find data structures that consume excessive amounts of memory. Work by Dufour *et al.* [9] uses a blended escape analysis to characterize and find excessive use of temporary data structures. By approximating object lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic areas. Shankar *et al.* propose Jolt [36], a tool that makes aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Recent work [26] finds memory leaks and bloat in C/C++ programs by segregating objects based on their allocation contexts and staleness. The research from [42] detects memory bloat by profiling copy chains. Our previous work proposes a container profiling approach to detecting Java memory leaks [44] and the work in [35] dynamically identifies inappropriately-used Java collections. The work from [43] proposes a dynamic technique to find low-utility data structures based on the profiling of cost and benefit. Our work differs from all these existing dynamic approaches in that it proposes the first static technique to find memory inefficiencies. It can be used both in the development phase to find mistakes and inappropriate coding patterns that may lead to bloat, and in the tuning phase, with the help of a dynamic allocation-frequency profile, to detect inefficiently-used containers that can be optimized across inputs, environments, and configurations.

CFL-reachability It is well known that method calls and returns can be treated as pairs of balanced parentheses using a context-free language [28, 30, 31, 32]. Sridharan and Bodik propose a CFL-reachability formulation to precisely model heap accesses and calling contexts for computing a points-to solution for Java [37]. As an extension of this formulation, Zheng and Rugina [46] define a CFL-reachability formulation for C/C++ alias analysis. Our previous work [45] proposes the CFL-reachability formulation of a must-not-alias analysis (based on a simplified context-free language) in order to scale the Sridharan-Bodik points-to analysis.

CFL-reachability can be used to implement polymorphic flow analysis [27], shape analysis [29], and information flow analysis [19]. The work in [16, 21] studies the connection between CFL-reachability and set-constraints, shows the similarity between the two problems, and provides new implementation strategies for problems that can be formulated in this manner. While based on the Sridharan-Bodik formulation of points-to analysis, our approach takes into account container-specific structures, and could potentially have a range of applications in problems that need reasoning about container behavior. Examples of such problems include summary generation for containers for more scalable static analyses, static container-based memory leak detection [13], and other techniques (e.g., thin slicing [38]) that need to track the flow of container elements while ignoring objects that form the internal container structures.

Object reachability analysis The work closest to ours is the disjoint reachability analysis proposed by Naik and Aiken [25] for eliminating false positives in their Java data race detector. This analysis is also flow-insensitive and takes into account loop information to distinguish instances created by the same allocation site. Unlike this analysis, which uses object-sensitivity to compute reachability information, we employ a CFL-reachability formulation that is capable of filtering out information irrelevant to container objects; as a result, our analysis may scale to larger programs. Other reachability analysis algorithms range from flow-sensitive approximations of heap shape (e.g., [6, 7, 34]) to decision procedures (e.g., [17, 20]). While our analysis is less precise in discovering the shape of data structures, it is more scalable and has been shown to be effective in detecting container problems.

There exists a large body of work on ownership types and their inference algorithms [1, 5, 8, 10, 13, 18]. Ownership types provide a way of specifying object encapsulation and enable local reasoning about program correctness in object-oriented languages. Existing ownership type inference algorithms may not be able to provide precise container information, because containers are usually designed to have polymorphic object ownership—some objects of a container type may own their elements while others may not. In addition, containers are complicated because they can be nested: the elements in a container may themselves be containers. As a pointer to a container is passed around, ownership of the container may transfer. While the work from [14] proposes an abstract object ownership model specifically for containers, it requires the tool users to specify correct interfaces for container implementation routines, which are then used in the ownership inference algorithm. Our approach is completely automated and does not require any user annotations for detecting problems with the built-in Java collections.

6. Conclusions

This paper presents practical static and dynamic analyses that can automatically find inefficiently-used containers. The goal of these analyses is to check, for each container, whether it has enough data added and whether it is looked up sufficient number of times. At the heart of these tools is a base static analysis that abstracts container functionality into basic operations *ADD* and *GET*, and detects them by formulating CFL-reachability problems. The identified operations can be used by a static inference engine that infers the relationship between their execution frequencies, and by a dynamic analysis that instruments these statements and finds bloat by profiling their frequencies. Experimental results show that the static analysis can scale to large Java applications and can generate precise warnings about the suspicious usage of containers. Promising initial case studies suggest that the proposed techniques could be useful for identifying container-related optimization opportunities.

Acknowledgments We thank Gary Sevitsky for his helpful comments on an earlier draft of this paper. We would also like to thank the anonymous reviewers for their valuable and thorough comments. This research was supported by the National Science Foundation under CAREER grant CCF-0546040 and by an IBM Software Quality Innovation Faculty Award.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330, 2002.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[3] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–72, 2006.

[4] M. D. Bond and K. S. McKinley. Leak pruning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009.

[5] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 213–223, 2003.

[6] C. Calcagno, D. Distefano, P. OHearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 289–300, 2009.

[7] B. Chang and X. Rival. Relational inductive shape analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 247–260, 2008.

[8] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310, 2002.

[9] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[10] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 241–255, 2001.

[11] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 375–385, 2009.

[12] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 127–139, 2009.

[13] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, 2003.

[14] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In *International Conference on Software Engineering (ICSE)*, pages 252–261, 2006.

[15] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007.

[16] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 207–218, 2004.

[17] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, and S. Srivastava. Simulating reachability using first-order logic with applications to verification of linked data structures. In *International Conference on Automated Deduction (CADE)*, pages 99–115, 2005.

[18] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *International Conference on Software Engineering (ICSE)*, pages 323–332, 2007.

[19] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 50–56, 2008.

[20] S. McPeak and G. Necula. Data structure specifications via local equality axioms. In *International Conference on Computer Aided Verification (CAV)*, pages 476–490, 2005.

[21] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248:29–98, 2000.

[22] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[23] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[24] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[25] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 327–338, 2007.

[26] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 397–407, 2009.

[27] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 54–66, 2001.

[28] T. Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction (CC)*, pages 389–403, 1994.

[29] T. Reps. Shape analysis as a generalized path problem. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 1–11, 1995.

[30] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[31] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[32] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 11–20, 1994.

[33] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, 2004.

[34] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 1999.

[35] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, 2009.

[36] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[37] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[38] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 112–122, 2007.

[39] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 59–76, 2005.

[40] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138, 2005.

[41] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

[42] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[43] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[44] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[45] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 98–122, 2009.

[46] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 197–208, 2008.