Dynamic Analysis of Inefficiently-Used Containers

Shengqian Yang[1] Dacong Yan[1] Guoqing Xu[2] Atanas Rountev[1]
[1]Ohio State University [2]University of California, Irvine

ABSTRACT

The goal of this work is to identify suspicious usage of containers, as an indicator of potential performance inefficiencies. To analyze container-related behavior and performance, we propose a dynamic analysis that tracks and records the flow of element objects to/from container objects. The observed interactions among containers and their elements is captured by a container-element flow graph. This graph is then analyzed by three detectors of potential container inefficiencies, based on certain patterns of suspicious behavior. In a promising initial study, this approach uncovered a number of performance problems in realistic Java applications.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms, measurement, performance

Keywords

Container, performance, inefficiency

1. INTRODUCTION

Object-oriented applications commonly suffer from significant performance problems. Experience shows that many such bottlenecks are caused by runtime bloat, a term referring to excessive memory usage and computation to accomplish simple tasks [7]. The underlying problem often comes from inappropriate design/implementation choices. Compilers typically cannot remove such bloat, because they lack understanding of the higher-level application semantics.

An important aspect of bloat is the inefficient use of containers [14, 10, 6]. Languages such as Java and C# provide numerous container data types, and they are heavily used by programmers. Containers can be misused in many ways. For example, each concrete implementation of an abstract data type may have different performance tradeoffs. Choosing the most appropriate implementation is challenging, and techniques exist to assist with this choice [10, 6]. Another typical inefficiency is the unnecessary invocation of container operations. For example, a container can be overpopulated if many elements are added to it, but only few of them are ever retrieved. While a static analysis has been proposed to detect some of these potentially-redundant operations [14], it does not consider interactions between containers, and cannot take advantage of the full wealth of run-time information in order to provide focused reports to the user.

A fundamental limitation of existing analyses of container inefficiencies is that they consider each container individually without modeling *interactions* between different containers (i.e., how objects flow from one container to another). However, understanding such interactions is necessary for developers to identify many types of container inefficiencies. For example, in our studies, we found a number of cases where an *intermediate container* is created only to transfer elements to/from other containers. Eliminating this intermediate container can sometimes result in a noticeable performance improvement.

Our Proposal. We propose a new technique to analyze the run-time behavior of containers, in relationship with other containers and the elements that flow between them. This approach provides a global view of the container-related behavior of the program. At the core of this dynamic analysis is the construction of a *container-element flow graph* (CEFG) in which nodes represent container objects and edges represent the flow of elements to/from containers. Nodes and edges are annotated with frequency information. Using this graph, various analyses can be developed to detect container-related inefficiencies. We describe three such detector analyses. The approach was implemented in the Jikes Research Virtual Machine [5]. Our initial experience is very promising: by inspecting the generated analysis reports, we found and fixed significant container-related performance problems in several Java applications.

2. REPRESENTING THE FLOW OF ELEMENTS AMONG CONTAINERS

2.1 Container-Element Flow Graph

Containers and Elements. A *container* is a data structure whose purpose is to store other objects (*element* objects), and to allow container clients to add and retrieve

WODA '12, July 15, 2012, Minneapolis, MN, USA

these elements. For our language-independent problem definition, a container object is an instance of some predefined abstract data type T. We assume that T provides operations whose semantics is to add elements to the container, as well as operations for retrieving these elements.

Basic Events. The run-time interactions of an element object e with containers can be represented by abstract operations $add(c, e)$ and $retrieve(c, e)$ where c is some container object. Other run-time behavior relevant to e can be abstracted as events $create(e)$, $use(e)$, and $assign(e)$. Event $create(e)$ represents the execution of the statement that creates e. Event $use(e)$ shows that e was used for some purpose unrelated to containers. The precise definition of "use" will depend on the language; for example, as discussed shortly, in Java one such use is the reading or writing of some field of e. Finally, event $assign(e)$ shows that (a reference to) e is assigned from one memory location to another. Examples include an assignment from one local variable to another, parameter passing, returning a reference from a call, and assignments from/to local variables to/from heap locations.

Flow Events. The lifetime of an element object e can be represented as a sequence $lifetime(e)$ of such events. This sequence can be used to define certain *flow events* which occur when an operation $add(c, e)$ is observed.

(1) Create-to-Add Flow. Flow $flow(e, c)$ occurs when element object e reaches container object c directly from the statement that created e, without going through any other containers. More precisely, in $lifetime(e)$, there must exist a subsequence of events starting with $create(e)$, ending with $add(c, e)$, and containing only $assign(e)$ intermediate events. Furthermore, any intermediate *assign*, as well as the final *add*, must obtain e from a memory location that is written by the previous *assign* in the subsequence (or, in the case of the first *assign*, by the *create*). This subsequence represents the flow of e through memory locations (on the stack and on the heap) due to assignments, calls, and returns.

(2) Retrieve-to-Add Flow. Flow event $flow(c_1, e, c_2)$ occurs if there exists a subsequence of $lifetime(e)$ starting with $retrieve(c_1, e)$, ending with $add(c_2, e)$, and containing only $assign(e)$ events. Any *assign*, and the final *add*, must obtain e from a location that is written by the previous *assign* in the subsequence (or by the *retrieve*, for the first *assign*). This event shows that e was in c_1, was retrieved from it, and from there reached c_2 without going in/out of another container. Similarly, $flow(c, e, use)$ occurs when e is retrieved from container c, and flows to a statement that uses e, without passing in/out of other containers along the way.

Consider an event $flow(c_1, e, c_2)$ starting with $retrieve(c_1, e)$ and ending with $add(c_2, e)$. An interesting case is when this flow is *pure*. Suppose that there exists an event $flow(c_1, e, use)$ which starts from the same $retrieve(c_1, e)$ and occurs before the moment of time when $add(c_2, e)$ occurs. This means that in the period of time between retrieving e from c_1 and adding e to c_2, e was not merely propagated, but actually served some useful purpose. In this case, $flow(c_1, e, c_2)$ does *not* represent pure flow. As discussed later, by identifying pure flow events, we can detect certain inefficiencies.

(3) Retrieve-to-Other Flow. It is possible that an element e obtained from a container c never flows into another container. To account for this, we introduce $flow(c, e, other)$ events. Suppose that for some $retrieve(c, e)$, there do not exist any $flow(c, e, c_2)$ starting from this retrieve event. Corresponding to each such $retrieve(c, e)$, we define a flow event $flow(c, e, other)$. Furthermore, if some $flow(c, e, use)$ starts from the same retrieve event, the retrieve-to-other flow is not pure. In cases where this flow is pure, this means that e was obtained from c but never used for anything.

Graph Nodes and Edges. The events described above serve as basis for the container-element flow graph. Every run-time object—element or container—is represented by a graph node. Various abstractions can be used, and our current implementation uses the standard approach of representing an object by the allocation site that created it. Each node n is annotated with a count $objs(n)$ of the number of objects represented by n. For every n representing container objects c, a count $adds(n)$ shows the number of $add(c, e)$ events. A count $retrieves(n)$ is defined similarly. For every container node n, a node $other(n)$ is added to represent events $flow(c, e, other)$, as described below.

If a node n_1 represents element objects e and a node n_2 represents container objects c, an edge from n_1 to n_2 indicates that $flow(e, c)$ occurred. The edge is annotated with the count $flows(n_1, n_2)$ of such flow events.

If n_1 represents container objects c_1 and n_2 represents container objects c_2, an edge from n_1 to n_2 shows that $flow(c_1, e, c_2)$ was observed for some e. The edge is annotated with the count $flows(n_1, n_2)$ of such flow events. Furthermore, another count $pure(n_1, n_2)$ shows how many of these events represent pure flow.

If a node n represents container object c, an edge from n to $other(n)$ shows that for some e, event $flow(c, e, other)$ was observed. The edge is annotated with the count of such flow events, as well as the number of them that are pure.

Detailed Example. The example in Figure 1 is derived from the `pmd` benchmark in the DaCapo suite [1, 2]. This example is based on a case study discussed in Section 4. Method `filterSource` is invoked to filter out certain elements in list `source` (created at line 3). First, the elements of `source` are added into an intermediate list `interim`. In order to avoid duplicated elements, a set `unique` of unique elements is maintained (and checked at line 19). After `interim` is constructed, `doFilter` selects only the even integers in it, adds them to another list (created at line 29), and returns this list back to `filterSource`. Finally, the elements after filtering are used to construct yet another list `afterFilter` which is used as the return value of `filterSource`.

The CEFG for this example is shown in Figure 2. Node n_i corresponds to the allocation site at line i in Figure 1. Count *objs* shows the number of objects created by this node. A container node n shows the total number of *add* and *retrieve* operations observed for objects represented by n. An edge is annotated with the number of corresponding events.

The edge from n_5 to n_3 shows the flow of one hundred `Integer` elements to the `source` list created at line 3. These objects then flow into n_{13} and n_{14} (`unique` and `interim`) due to the loop at lines 17–23. As part of this loop, every element of `source` is used as a parameter in a call to `contains` (line 19). For the purpose of the example, assume that this method uses the value of the parameter to perform the membership test. This use of the elements of `source` means that the flows from n_3 to n_{13} and n_{14} are not pure, which is reflected by counters *pure* on the corresponding edges.

Inside `doFilter`, the hundred elements of n_{14} are used in a call to `intValue`, which is represented by counters *pure* on the edges to $other(n_{14})$ and n_{29}. Fifty of them flow to `res`, represented by n_{29}; the remaining fifty do not flow into

```
1   static List source;
2   public static void main(String []args) {
3      source = new ArrayList();
4      for (int i = 0 ; i < 100 ; i++) {
5        Integer integer = new Integer(i);
6        source.add(integer);
7      }
8      List result = filterSource();
9      System.out.println(result.toString());
10  }
11  public static List filterSource() {
12     Filter filter = new Filter();
13     Set unique = new HashSet();
14     List interim = new ArrayList(source.size());
15     List afterFilter = new ArrayList();
16     Iterator iterator = source.iterator();
17     while (iterator.hasNext()) {
18       Object eachAxisNode = iterator.next();
19       if (!unique.contains(eachAxisNode)) {
20          unique.add(eachAxisNode);
21          interim.add(eachAxisNode);
22       }
23     }
24     afterFilter.addAll(filter.doFilter(interim));
25     return afterFilter;
26  }
27  class Filter {
28  public List doFilter(List src) {
29     List res = new ArrayList(src.size());
30     Iterator iterator = src.iterator();
31     while (iterator.hasNext()) {
32       Object element = iterator.next();
33       if ((Integer)element).intValue() % 2 == 0) {
34            res.add(element);
35       }
36     }
37     return res;
38   }
39  }
```

Figure 1: Detailed example.

any containers, which corresponds to the edge from n_{14} to $other(n_{14})$. Due to the call to `addAll` at line 24, the fifty elements from n_{29} flow to list `afterFilter` (node n_{15}), and this flow is pure, as indicated by the counters on the edge to n_{15}. The edge from n_{15} to $other(n_{15})$ is due to the call to `toString` at line 9; for illustration purposes, we assume that this method retrieves all elements of n_{15} and uses them.

In this example, the pure flow from n_{29} to n_{15} presents a typical case of inefficiency. List `afterFilter` is used as an intermediate container with no benefit to the execution, and can be eliminated (i.e., `filterSource` can directly return the result of the call to `doFilter` at line 24). In our study of the actual `pmd` application, we found an even more interesting problem. The filtering is based on a set of filters, and this set is often empty—that is, all elements of `interim` (n_{14}) have pure flow to `afterFilter` (n_{15}). Creating specialized code for this case can lead to substantial improvements.

2.2 Building the Graph for Java Programs

Containers and Their Operations. We chose to focus on several widely-used container types in the Java libraries that implement interfaces `List`, `Set`, and `Map`. Calls to operations from these types are mapped to *add* and *retrieve* events. This mapping is straightforward, but a few operations require more complex treatment. For example, the call to `addAll` in Figure 1 is considered to be a sequence of $retrieve(c_1, e)$ events, for all elements e of the container c_1 returned by `doFilter`, followed by a sequence of $add(c_2, e)$ where c_2 is the container referenced by `afterFilter`.

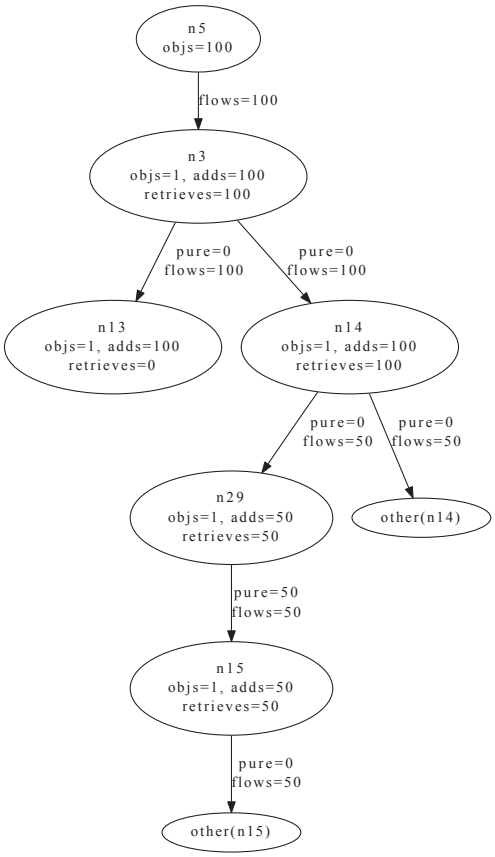

Figure 2: Flow graph for the `pmd` example.

Iterators in Java can be used to access or modify a container. For the container types we analyze, there are several related iterator types, and operations in these types are accounted for by the analysis. For example, the call to `next` at line 18 in Figure 1 is equivalent to a $retrieve(c, e)$ event for the underlying container c.

Tracking the Flow of Element Objects. A $create(e)$ event is simply the execution of the allocation site that creates e. Any subsequent assignments in which a (reference to) e is assigned from one memory location to another are tracked. The run-time dependences between such $assign(e)$ events (via the memory locations that store the references to e) are also tracked. This is needed to identify occurrences of events $flow(e, c)$, $flow(c_1, e, c_2)$, and $flow(c, e, use)$ based on the definitions from Section 2.1.

To identify occurrences of $use(e)$ events, we consider statements that make use of a variable `v` that references object e. For example, field accesses via `v.fld` and calls via `v.m()` are considered uses of e. Other examples include array accesses `v[i]`, expressions `v instanceof X`, comparisons `v==v2` or `v!=v2`, and uses of `v` as a parameter of native calls.

Whenever *create*, *add*, or *retrieve* is observed, the related counter for the corresponding graph node is incremented. In addition, for an $add(c, e)$, the tracking information is used to determine the source of e (its allocation site, or another container) and whether e has been used since the start of this flow. The *flows* and *pure* counters on the graph edge are incremented correspondingly. Additional counters are used to account for elements that have been retrieved but have not yet reached another container, in order to record $flow(c, e, other)$ events and their purity.

2.3 Analysis Implementation

The proposed analysis for Java programs is implemented in Jikes RVM 3.1.1 [5], using techniques similar to existing work [16, 12]. The decision to implement the analysis inside a virtual machine was made for convenience of implementation; the technique can be easily implemented with offline bytecode instrumentation instead.

Shadow Locations. Each memory location containing an object reference could potentially refer to an element object. To track the source of this object, each location has a corresponding shadow location. The shadow stores a reference to a tracker object, which itself contains (1) a reference to the CEFG node for the source of the element object, and (2) a boolean flag for tracking *use* events. The CEFG node is either the node for the allocation site that created the element object, or the node for the allocation site of the container from which this element object was retrieved last.

For local variables, the shadows are created by adding symbolic registers to the internal representation used by the VM compiler. For instance fields, shadow locations are added inside the corresponding objects. For static fields and array elements, various tables are used, and indices into these tables are based on the field ID or the array index.

Run-time Construction of the CEFG. Each container allocation site is assigned a unique ID at instrumentation time, and this ID is written in the header of each container. Upon $add(c, e)$ or $retrieve(c, e)$, the ID stored in c is used to find the corresponding graph node. When $retrieve(c, e)$ occurs, some memory location x starts referring to the retrieved element e. A reference to a tracker object (which itself refers to the graph node for c) is written to the shadow for x. Subsequent assignments $y = x$ copy the shadow of x into the shadow of y. When the reference to e flows into an *add* event, the shadows are used to determine the graph node corresponding to the container from which e was retrieved, and $flow(c_1, e, c_2)$ is recorded. Events $flow(c, e, use)$ are detected in a similar fashion, and bookkeeping for them is done through a boolean flag inside the tracker object. To handle containers and elements accessed by multiple threads, locks are used on tracker objects and CEFG elements.

Run-time Overhead. Although our prototype implementation is not optimized in any significant way, it exhibits practical cost. The median running time overhead is 9.6×, and the median memory overhead is 2.0×. These overheads are lower than those for similar analyses with fine-grain flow tracking (e.g., [16, 12]), and are acceptable for performance tuning and debugging tasks.

3. DETECTING POTENTIAL CONTAINER INEFFICIENCIES

The CEFG could be used as the basis for *detector analyses* to identify containers that may be inefficiently used. To illustrate this approach, this section describes three such analyses, but additional detectors could be developed in future work. Each detector defines graph-based metrics to find and rank suspicious containers. The three detectors are parameterized by a threshold value for their metrics; a value of 0.5 was used in all our experiments.

Detection of Intermediate Containers. Consider the case when element objects e exhibit pure flow from one container c_1 to another container c_2. This is a potentially interesting case because c_1 or c_2 may be considered to be an intermediate container whose existence is primarily motivated by the need to replicate information available in the other container, without actually using this information for something useful. For example, in Figure 2, elements are retrieved from n_{29} and added to n_{15} without any uses. In this case we can eliminate the creation of either n_{29} or n_{15}.

This analysis can be implemented as follows. For any CEFG edge connecting two container nodes, we consider the ratio $(flows - pure)/flows$. If this ratio is small (i.e., close to zero), most of the flow is pure, which raises the following question: if the flow is primarily used to populate the target container, and not to perform useful work, is the existence of both containers justified? When this ratio is smaller than a predefined threshold, the edge is reported as suspicious. The detector produces a ranked list of suspicious edges, ordered by the value of the *flows* counter.

A special case of this analysis are edges from a container node n to the corresponding node $other(n)$. When the flow along such an edge is mostly pure, most elements retrieved from the container are never used and also are never added to other containers, which may be a symptom of inefficiencies.

Detection of Underutilized Containers. This detector analysis identifies container allocation sites that allocate a large number of container objects, while very few element objects are added to them. These containers are *underutilized* in that they are created for the purpose of storing element objects, but end up storing only a very small number of them. Such containers consume more memory than strictly necessary, due to the overhead of maintaining internal container data structures [8]. The cost of building and then garbage collecting these data structures can sometimes be problematic, and this cost may not always be justified.

For each container node n, the detector computes the ratio $adds(n)/objs(n)$. If this ratio is smaller than the predefined threshold, n is reported as a producer of potentially underutilized container objects. The reported nodes are ranked by the number $objs(n)$ of allocated objects.

Detection of Overpopulated Containers. Consider a different scenario, where a large number of elements is added to a container—i.e., it is utilized substantially. However, there can still be inefficiency associated with this container if the program performs only a small number of *retrieve* operations (e.g., smaller than the number of *add* operations) [14]. The very reason the program stores objects in containers is to allow container clients to retrieve these elements. The fewer times such retrievals occur, the less justified the work and memory needed to populate a container.

This analysis computes, for a container node n, the ratio $retrieves(n)/adds(n)$. When the ratio is smaller than a threshold, the containers represented by n are considered to be potentially overpopulated. The analysis ignores nodes n with $adds(n) = 0$, since they are clearly underutilized. The reports are ranked by the number of *add* operations.

4. CASE STUDIES

To understand the suspicious containers identified by the detectors, we performed case studies on Java programs used in [16, 15]. Various interesting patterns of container-related inefficiencies were detected. For illustration, we discuss a subset of these case studies that exhibit a diversity of runtime symptoms. With one exception, the problems described below have not been reported by any prior work. The extent and variety of these container-related inefficiencies provide

strong motivation for performing dynamic analyses of containers (e.g., based on the CEFG or similar abstractions).

pmd. In the `pmd` benchmark from DaCapo [1, 2], there exists a performance problem related to intermediate containers. The example in Figure 1 is based on this problem. In class `DefaultStep`, two `ArrayList`s are created every time `evaluate` is called. The first list is populated with elements and is then transferred to a method which filters certain unnecessary elements. After filtering, all remaining elements are added to the second list. The filtering is performed based on a set of filters. However, sometimes this set does not contain any filters; thus, all elements added to the first list flow to the second one, and this flow is pure. The detector of intermediate containers (Section 3) reports this problem as the top-ranked suspicious graph edge. There is a simple specialization to address this inefficiency. A similar problem reported by the detector is due to flow during which a non-empty set of filters is applied to an often-empty set.

These examples illustrate the importance of specializing for the "simple" case which can be performed efficiently. While such specialization is not advisable during development, when functionality and correctness are the primary concerns, it becomes important during performance tuning.

Another interesting example from this benchmark is related to overpopulated containers. The top-ranked allocation site reported by the detector of overpopulated containers is for a set created in class `IdentitySet`. A large number of elements are added to this set, but are never retrieved from it. The sets are used to answer membership queries. To implement the desired semantics, both the set elements and the query objects are wrapped in wrapper objects before being processed with the set. Although fresh wrapper objects are created for each query, it is easy to see that these wrappers can be reused across queries. This simple optimization eliminates about three million run-time objects.

The changes described above are semantics-preserving and very easy to implement. After these changes, the running time of the application is reduced by 13.6% and the number of created objects is reduced by 12.5%.

lusearch. In this benchmark from DaCapo, we examined the top-ranked underutilized containers and focused on two of them that were highly suspicious because no elements were ever added to them. These are extreme cases of underutilized containers, because the cost of creating and garbage collecting them is not justified by any actual use. One such container is created in class `StandardTokenizer` and is used to store parse exceptions. In cases when no parse exceptions occur, this container remains empty. A similar situation is observed for an allocation site in class `QueryParser`. The natural solution is to create these containers on demand rather than up front. This is another example of specialization that can be introduced during performance tuning, when the cost of up front container creation can be quantified by our analysis, and the programmer's effort to introduce a specialized on-demand solution can be justified.

The top-ranked overpopulated container is a `Vector` created in class `QueryParser`. This container is used to store the clauses that form a search query. In the case when there is a single clause in the query, the container is still populated with this clause, but nothing is retrieved: in the code, this one clause is returned directly, which makes the container redundant. The creation of the container can be postponed until there is an attempt to add a second clause; at this time, both the first and the second clause are added. This change and the changes described earlier reduce the running time by 4.8% and the number of created objects by 3.4%.

bloat. For the `bloat` benchmark from DaCapo, the top-ranked overpopulated container is a set created by an allocation site in class `RegisterAllocator`. This set is used to compute the union of two existing sets. Once populated, the union set is used only to compute the size of the union, and its elements are not retrieved. However, there is a more efficient way to compute this size. A counter can be initialized to the size of the larger of the two given sets. Iteration over the smaller set can check each element for membership in the larger set, and the counter can be incremented appropriately. Of course, it is easier for the programmer to write code that explicitly constructs the union set and simply calls `size()` on it, but this convenience comes with a performance penalty. This way of thinking is not uncommon for Java programmers: the container types are easily available in the standard libraries, and it is tempting to think of them as primitive types, with little or no consideration for the cost of their operations. In fact, this way of thinking is often encouraged by the way Java (and similar languages) are taught to students, where emphasis is put on code simplicity and reuse. However, performance considerations will have to come into the picture sooner or later.

The detector of intermediate containers highlighted another problem with this computation. Based on the reports, it became clear that the elements of the two sets being unioned can flow only to the union set, and this flow is pure. In fact, these sets are created only to serve as wrappers around existing sets, and they are redundant when the size of the union is being computed. They can be easily eliminated without affecting program semantics. By rethinking the implementation and introducing these two rather simple optimizations, we achieved 15.6% reduction in running time and 14.0% reduction in the number of allocated objects.

ps. In this DaCapo benchmark, the detector of intermediate containers reports a large number of potentially problematic nodes. We examined the five top-ranked nodes, and determined that all of them are due to improper use of stack containers. As one example, a computation that determines the stack size is implemented by transferring the elements to a temporary stack, and then transferring them back to the original stack. As another example, similar problems are observed when an element needs to be retrieved from a particular position in the stack, although the stack API already provides an efficient way to achieve this. Accidentally, some of these problems have been identified in [15] by an unrelated static analysis. However, that analysis is not designed specifically to address container-related inefficiencies, and identifies another symptom of the underlying problem. Our dynamic detector of intermediate containers strongly suggests that the element flow between the stack containers is suspicious. After eliminating the inefficiencies related to these five container allocation sites, the running time of the benchmark is reduced by 13.5% and the number of allocated objects is reduced by 56.0%.

soot. Soot, a Java optimization framework, extensively uses containers. We ran soot-2.2.3 using as input the bytecode of the antlr DaCapo benchmark. Among the top five reports from the detector of intermediate containers, three allocation sites in class `TypeVariable` stood out. Upon examination, it became clear that there is a substantial amount

of data transfer between containers. A `LinkedList` is used to store the parents of a type variable. This information is needed as part of a type analysis of local variables. In several places such a list is used to create an intermediate `TreeSet`, which is then updated by adding or removing elements, and finally translated back into another `LinkedList` which is remembered as the collection of parents. It is not clear that it is necessary to use both lists and sets for this functionality. We changed the code to use and remember only sets, while being careful to preserve the semantics.

A similar case exists in class `SmartLocal`, where the elements of a list are purely copied into a set to ensure that each element is unique. This set is not modified further and is used only to check membership. By addressing these problems related to intermediate containers, the modified program shows 4.0% reduction in running time and 3.1% reduction in the number of allocated objects.

5. RELATED WORK

Detecting Container-Related Inefficiencies. The work in [4] and [13] focuses on container usage in order to detect memory leaks. Some dynamic analyses make container selection recommendations: Shacham et al. [10] apply a set of rules to adaptively select Java containers for memory improvements, while Jung et al. [6] combine profiling and machine learning to take into account the architecture and the input data when making suggestions. The work in [14] uses a static analysis to infer container semantics and report suspicious containers. However, applications of this analysis are limited to detecting underutilized and overpopulated containers, and do not consider interactions between containers. In addition, it is impossible to use this technique to find hot inefficiently-used containers (i.e., containers that are more likely to be bottlenecks), which is a typical problem for all static analyses of performance problems.

As described in Section 1, this existing work focuses on the usage of a single container, and thus may miss opportunities of detecting inefficiencies that involve multiple containers, or the flow of elements outside of containers. In addition, the CEFG proves a unified view that could be used for various other analyses of container-related performance problems.

Detecting Run-time Bloat. Mitchell et al. [9] analyze the program data flow to detect bloat, and their later work [8] computes health signatures to locate memory-demanding data structures. Dufour et al. [3] use a blended escape analysis to find excessive use of temporary data structures. Jolt [11] identifies program regions that make extensive use of temporary objects. Xu et al. [12] employ abstract slicing to find low-utility data structures based on the numbers of instructions executed to produce and use a run-time value. The work in [16] profiles the program to track reference propagation in order to uncover low-benefit object allocations.

Our work differs from all these existing approaches in that it leverages the higher-level container semantics, tracks interactions between multiple containers, and specifically targets container-related performance inefficiencies. This allows programmers to quickly find and remove bloat by focusing on this important category of inefficiencies.

6. CONCLUSIONS AND FUTURE WORK

We propose to capture the flow of element objects across container boundaries, in order to identify performance problems related to container behavior. This flow is represented by a container-element flow graph. This graph, in its current form or with additional metrics for nodes/edges, could serve as the basis for various analyses of container usage. We propose three such analyses, and discuss case studies in which interesting container inefficiency patterns are uncovered by them. Significant improvements can be achieved by manually optimizing the reported suspicious containers.

These initial studies are promising, as they suggest that container-focused optimization is a legitimate target for performance analysis and tuning. It is important to raise the awareness of the programmer about the cost of seemingly innocent use of well-known container types. Although it may be convenient to think of such types as if they were primitive types, the performance implications of such thinking cannot be ignored. The proposed approach can assist programmers, both through the detectors presented here as well as other detectors developed in future work, to detect and eliminate container-related performance problems.

Acknowledgments. We thank the WODA reviewers for their feedback. This material is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040, grant CCF-1017204, and by an IBM Software Quality Innovation Faculty Award. Guoqing Xu was supported in part by an IBM Ph.D. Fellowship Award.

7. REFERENCES

[1] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.
[2] DaCapo Benchmarks, `www.dacapo-bench.org`.
[3] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *FSE*, 2008.
[4] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In *ICSE*, 2006.
[5] Jikes RVM, `jikesrvm.org`.
[6] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *PLDI*, 2011.
[7] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1), 2010.
[8] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, 2007.
[9] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *ECOOP*, 2006.
[10] O. Shacham, M. Vechev, and E. Yahav. Chameleon: adaptive selection of collections. In *PLDI*, 2009.
[11] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *OOPSLA*, 2008.
[12] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, 2010.
[13] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE*, 2008.
[14] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, 2010.
[15] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *ECOOP*, 2012.
[16] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *ICSE*, 2012.