Distinguished Paper

Precise Memory Leak Detection for Java Software Using Container Profiling*

Guoqing Xu Atanas Rountev
Department of Computer Science and Engineering
Ohio State University
{xug,rountev}@cse.ohio-state.edu

ABSTRACT

A memory leak in a Java program occurs when object references that are no longer needed are unnecessarily maintained. Such leaks are difficult to understand because static analyses typically cannot precisely identify these redundant references, and existing dynamic analyses for leak detection track and report fine-grained information about individual objects, producing results that are usually hard to interpret and lack precision.

We introduce a novel *container-based* heap-tracking technique, based on the observation that many memory leaks in Java programs occur due to containers that keep references to unused data entries. The novelty of the described work is two-fold: (1) instead of tracking arbitrary objects and finding leaks by analyzing references to unused objects, the technique tracks only containers and directly identifies the source of the leak, and (2) the approach computes a confidence value for each container based on a combination of its memory consumption and its elements' staleness (time since last retrieval), while previous approaches do not consider such combined metrics. Our experimental results show that the reports generated by the proposed technique can be very precise: for two bugs reported by Sun and for a known bug in SPECjbb, the top containers in the reports include the containers that leak memory.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Reliability, Performance, Measurement, Experimentation

Keywords

Memory leaks, container profiling, leaking confidence

*This material is based upon work supported by the National Science Foundation under grant CCF-0546040.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. INTRODUCTION

While garbage-collected languages can reduce memory-related bugs such as dangling pointers, programs written in these languages can still suffer from memory leaks caused by keeping references to useless objects. Leaks degrade runtime performance and significant leaks even cause the program to run out of memory and crash. In addition, memory leak bugs are notoriously difficult to find. Static analyses can be used to attempt the detection of such leaks. However, this detection is limited by the lack of scalable and precise reference/heap modeling, as well as by reflection, multiple threads, scalability for large programs, etc. Thus, in practice, identification of memory leaks is more often attempted with dynamic analyses. Existing dynamic approaches for heap diagnosis have serious limitations. Commercial tools such as JProfiler [13] and JProbe [12] were developed to help understand types, instances, and memory usage. However, this information is insufficient for programmers to locate a bug. For example, in most cases, the fact that type `java.util.HashMap$Entry` has the highest number of instances tells the programmer nothing about the hash maps that hold these entries. Research tools for memory leak detection typically focus on heap differencing [3, 4, 14] and fine-grained object tracking [1, 8, 7, 20].

Of existing dynamic techniques, LeakBot [17], Cork [14], and Sleigh [1] represent the state of the art. Both LeakBot and Cork use heap growth as a heuristic, which could result in false positives (growing types are not necessarily true leaks). Sleigh, on the other hand, uses staleness (time since last use) to find leaks. This approach could lead to imprecision as well. As an example, a frame in a Java Swing program cannot be treated as a leak, although it may never be used after it is created. In addition, larger objects that are less stale may have greater contribution towards the leak. For example, more attention should be paid to a big container that is not used for a while than to a never-used string. Furthermore, these existing tools follow a traditional *from-symptom-to-cause* approach that starts from tracking all objects and finds those that could potentially be useless (symptom). It then tries to find the leaking data structure (cause) by analyzing direct and transitive references to these useless objects. However, the complex run-time reference relationships among objects in modern Java software significantly increases the difficulty of locating the source of the leak, which could lead to imprecise leak reports. It becomes even harder to find the cause of a leak if there are multiple data structures that are contributing to the problem. For example, as reported in [14], it took the authors a

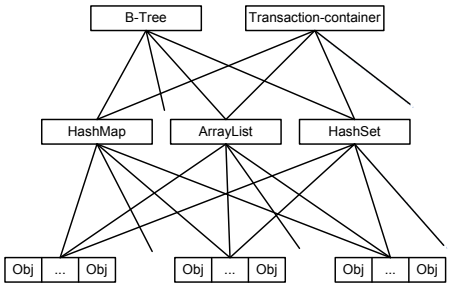

Figure 1: Container hierarchy in Java.

significant amount of time to find the sources of leaks after they examined the reports generated by Cork.

Our proposal. We propose a novel technique for Java that detects memory leaks using container profiling. Arguably, misuse of (user-defined or Java built-in) containers is a major source of memory leak bugs in real-world Java applications. For example, many of the memory leak bugs reported in the Sun bug repository [23] were caused, directly or indirectly, by inappropriate use of containers. The key idea behind the proposed technique is to *track operations on containers rather than on arbitrary objects, and to report containers that are most likely to leak.* The major difference between our technique and the from-symptom-to-cause diagnosis approach is that we start by suspecting that all containers are leaking, and then use the "symptoms" to rule out most of them. Hence, we avoid the process of symptom-to-cause searching that can lead to imprecision and reduced programmer productivity.

Figure 1 shows the container hierarchy typically used in a Java program: user-defined containers in the top layer use containers provided by the Java collection framework (the middle layer), which eventually store data in arrays (the bottom layer). The focus of our technique are containers in the first and second layers, because in most cases these containers are directly manipulated by programmers and hence more likely to be sources of leaks. Arrays are not tracked; completementary approaches such as [21] can potentially be used to detect leaks directly caused by arrays.

Our technique requires ahead-of-time modeling of containers: users need to build a simple "glue layer" that maps methods of each container type to primitive operations (e.g., ADD, GET, and REMOVE). An automated tool instruments the application code and uses the user annotations to connect invocations of container methods with our run-time profiling libraries. To write this glue code, users have to be familiar with the container types used in the program. This does not increase the burden on the programmers: when using existing leak detection tools [17, 1, 14], programmers have to inspect the code to gain similar knowledge so that they can interpret the tool reports. Using our approach requires learning such knowledge in advance. Of course, the tool embeds pre-defined models for containers from the Java collection framework, and therefore programmers need to model only user-defined containers. Running the tool even without modeling of user-defined containers can still provide useful insights for finding leaks: in our reports, top-level Java library containers (i.e., the second layer in Figure 1) can direct one's attention to their direct or transitive owners, which are likely to be user-defined containers (i.e., the first layer in Figure 1) that are the actual causes of bugs.

We compute a heuristic *leaking confidence* value for each container, based on a combination of its memory consumption and the staleness of its data elements; this could yield more accurate results compared to existing approaches [17, 1, 14]. For each container, we also rank *call sites* in the source code, based on the average staleness of the elements retrieved or added at these sites. This container ranking and the related call site ranking can assist a programmer to quickly identify the source of the memory leak. The conceptual model used to compute these values and our implementation of the technique for Java are presented in Section 2 and Section 3, respectively. Our tool achieved high precision in reporting causes for two memory leak bugs from the Sun bug database [23] and a known memory leak bug in SPECjbb [22] — in fact, the top containers in the reports included the ones that leaked memory. In addition, an evaluation of the run-time performance showed acceptable overhead for practical use.

Contributions. The main contributions of this work are:

- A dynamic analysis that computes a confidence value for each container, providing a basis for identification and ranking of likely-leaking containers
- A memory leak detection technique for Java based on this confidence analysis
- A tool that implements the proposed technique
- An experimental study of leak identification and runtime performance, showing that our technique can precisely detect memory leak bugs with practical overhead

2. LEAK CONFIDENCE ANALYSIS

This section presents a confidence analysis that computes leaking confidence values for tracked containers. The goal of the analysis is to quantify the contribution of a container to memory leaks.

A *container* is an abstract data type (ADT) with a set of data elements and three basic operations ADD, GET, and REMOVE. We use σ^n to denote a container σ with n elements. The simplified effects of the operations are as follows (o denotes a container element):
ADD$(\sigma^n, o) : void \equiv \sigma^n_{pre} \rightarrow \sigma^{n+1}_{post}$, $o \notin \sigma^n_{pre}$, $o \in \sigma^{n+1}_{post}$
GET$(\sigma^n) : o \equiv \sigma^n_{pre} = \sigma^n_{post}$, $o \in \sigma^n_{pre}$
REMOVE$(\sigma^n, o) : void \equiv \sigma^n_{pre} \rightarrow \sigma^{n-1}_{post}$, $o \in \sigma^n_{pre}$, $o \notin \sigma^{n-1}_{post}$

We treat all (Java library and user-defined) containers as implementations of the container ADT. Tracking operations on a container requires user-supplied annotations to bridge the gap between methods defined in the Java implementations and the three basic ADT operations. We have already defined such annotations for the container types from the standard Java libraries.

During the execution of a program, let the program's memory consumption at a timestamp τ_i be m_i. In cases when τ_i is a moment immediately after garbage collection (we will refer to such moments as *gc-events*), it will be denoted by τ^{gc}_i and its memory consumption will be denoted by m^{gc}_i. A program written in a garbage-collected language has a *memory leak symptom* within a time region $[\tau_s, \tau_e]$ if (1) for every gc-event τ^{gc}_i in the region, $m_s \leq m^{gc}_i \leq m_e$, and (2) in this region, there exists a subsequence $ss = (\tau^{gc}_1, \tau^{gc}_2, \ldots, \tau^{gc}_n)$ of gc-events, with $n \geq 2$, such that $\tau^{gc}_j < \tau^{gc}_{j+1}$ and $m^{gc}_j < m^{gc}_{j+1}$ for $j = 1, \ldots, n-1$. The period $[\tau_s, \tau_e]$ will be referred to as a *leaking region*.

This definition helps to identify the appropriate time region to analyze, because most programs do not leak from the start of execution. End moment τ_e can be specified by tool users as an analysis parameter, and can be different for different kinds of analyses. For post-mortem off-line diagnosis, τ_e is either the ending time of the program, or the time when an OutOfMemory error occurred. For on-line diagnosis done while the program is running, τ_e could be any time at which the user desires to stop data collection and to start analysis of the collected data. We use gc-events as "checkpoints" because at these times the program's heap memory consumption does not include unreachable objects.

The definition of a memory leak symptom does not require the amount of consumed memory at each gc-event to be larger than it was at the previous one, because in many cases some gc-events reclaim large amounts of memory, while in general the memory footprint still keeps increasing. The ratio between the number of elements n in the subsequence ss and the size of the entire sequence of gc-events within the leaking region can be defined by tool users as another analysis parameter, in order to control the length of the leaking region. Given a particular value for this user-defined ratio, there could be multiple values of τ_s corresponding to this ratio. Our approach chooses the smallest such value as τ_s, which defines the longest region and allows more precise analysis. (Additional details are described in Section 3.)

A container σ is *memory-leak-free* if either (1) at time τ_e, it is in state σ^0 (i.e., empty), or (2) it is garbage collected within the leaking region. That is, σ^n does not leak memory if at time τ_e, its accumulated number of ADD operations is equal to its accumulated number of REMOVE operations, assuming we treat the deallocation of σ^n as being equivalent to n REMOVE operations. Containers that are not memory-leak-free contribute to the memory leak symptom and are subject to further evaluations. However, this does not necessarily mean that all of them leak memory. For example, if an OutOfMemory error occurs before some REMOVE operations of a container, this container is not memory-leak-free according to the above definition, although in reality it may very well be leak-free.

For each container that is not memory-leak-free by this definition, we compute a confidence value that indicates how large is its contribution to the memory leak symptom. Our technique considers both the memory consumption and the staleness when computing the confidence for a container.

Memory contribution. One factor that characterizes a container's contribution to the leak is the amount of memory the container consumes during its lifetime. We quantify this factor by defining a *memory time graph* which captures a container's memory footprint.

The relative memory consumption of a container σ at time τ is the ratio between the sum of the memory consumption of all objects reachable from σ in its object graph, and the total amount of memory consumed by the program at τ. The memory time graph for σ is a curve where the x-axis represents the relative time of program execution (i.e., τ_i/τ_e for timestamp τ_i) and the y-axis represents the relative memory consumption of σ (i.e., $mem(\sigma)_i/total_i$ corresponding to x-point τ_i/τ_e). The starting point of the x-axis is τ_0/τ_e where $\tau_0 = max(\tau_s$, allocation time of $\sigma)$ and the ending point is τ_1/τ_e where $\tau_1 = min(\tau_e$, deallocation time of $\sigma)$.

A sample graph is shown in Figure 2. The x-axis starts at 0.4 relative time ($0.4 \times \tau_e$ absolute time), which represents

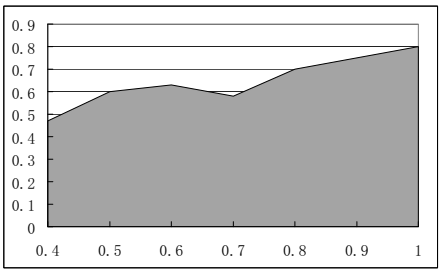

Figure 2: A sample memory time graph.

either the starting time of the leak region τ_s or σ's allocation time, whichever occurs later. The graph indicates that σ is not freed within the leak region, because the x-axis ends at 1, which represents the ending time τ_e of the leak region.

Using the memory time graph, a container's *memory contribution* (MC) is defined to be the area covered by the memory consumption curve in the graph. In Figure 2 this area is shown in dark. Because the graph starts from τ_s (or later), the MC considers only a container's memory consumption within the leaking region. For a container, both its memory consumption and its lifetime contribute to its MC. Since MC should reflect the influence of both the container itself and all objects (directly or transitively) referenced by it, the memory consumption of the container is defined as the amount of memory consumed by its entire object graph.

Because relative values (i.e., between 0 and 1) are used to measure the memory consumption and the execution time, the MC of a container is also a value between 0 and 1. Containers that have larger MC contribute more to the memory leak symptom. Note that in practice it is likely to be too expensive to compute the exact MC value for a container, because the container's memory consumption changes frequently as the program executes. Section 3 presents a sampling approach that can be used to approximate this value.

Staleness contribution. The second factor that characterizes a container's contribution is the staleness of its elements. The staleness of an object is defined in [1] as the time since the object's last use. We provide a new definition of staleness in terms of a container and its elements.

The *staleness* of an element o in a container σ is $\tau_2 - \tau_1$ where REMOVE(σ, o) occured at τ_2, an operation GET(σ):o or ADD(σ, o) occured at τ_1, and there does not exist another GET operation that returns o in the region $[\tau_1, \tau_2]$. If $\tau_1 < \tau_s$, τ_1 is redefined to be τ_s. If $\tau_2 < \tau_s$, the staleness is undefined. In other words, the staleness of o is the distance between the time when o is removed from σ and the most recent time when o is retrieved from σ. If o is never retrieved from σ, τ_1 should correspond to the ADD operation that adds o to σ. If o is never removed from σ, τ_2 is either the deallocation time of σ, or the ending time of the leaking region τ_e. The intuition behind this definition is that if the program no longer needs to retrieve an element from a container, the element becomes useless to that container. Hence, the staleness of the element measures the period of time when the element becomes useless but is still being kept by the container. In addition, tracking occurs only within the leaking region — if an element's removal time τ_2 is earlier than the starting time of the leaking region, we do not compute the staleness for the element.

The *staleness contribution* (SC) of a container σ is the ratio of $(\sum_{i=1}^{n} staleness(o_i)/n)$ and $(\tau_e - \tau_s)$, where the sum

ID	*Type*	*LC*	*MC*	*SC*
11324773	util.HashMap	0.449	0.824	0.495
18429817	util.LinkedList	0.165	0.820	0.194
8984226	util.LinkedList	0.050	0.809	0.062
2263554	util.WeakHashMap	0.028	0.820	0.034
15378471	util.LinkedList	0.018	0.029	0.256
5192610	swing.JLayeredPane	0.011	0.824	0.013
30675736	swing.JPanel	0.011	0.824	0.013
19526581	swing.JRootPane	0.011	0.824	0.013
17933228	util.Hashtable	0.000023	0.0007	0.026

Table 1: Partial report of LC, MC, and SC values.

is over all elements $o_1, \ldots, o_n$ that have been added to σ and whose staleness is well-defined. Thus, SC is the average staleness of elements that have ever been added to σ, relative to the length of the leaking region. In addition, the removal time of these elements must be within the leaking region. Because the staleness of each individual element is $\leq$ the length of the leaking region, SC is a value between 0 and 1. Containers that have larger SC values contribute more to the memory leak symptom.

Putting it all together: leaking confidence. Based on the memory contribution and the staleness contribution, we define a container's *leaking confidence* (LC) to be computed as $\text{SC} \times \text{MC}^{1-\text{SC}}$. Clearly, LC is a value between 0 and 1; also, increasing either SC or MC while keeping the other factor unchanged increases LC. We define LC as an exponential function of SC to show that staleness is more important than memory consumption in determining a memory leak. This definition of LC has several desirable properties:

- MC=0 and SC$\in$[0, 1] $\Rightarrow$ LC=0. If the memory contribution of a container is small enough (i.e., close to 0), the confidence of this container is close to 0, no matter how stale its elements are. This property helps filter out containers that hold small objects, such as strings.
- SC=0 and MC$\in$[0, 1] $\Rightarrow$ LC=0. If every element in a container gets removed immediately after it is no longer used (i.e., the time between the GET and REMOVE operations is close to 0), the confidence of this container is 0, no matter how large the container is.
- SC=1 and MC$\in$[0, 1] $\Rightarrow$ LC=1. If all elements of a container never get removed after they are added (i.e., every element crosses the entire leaking region), the confidence of the container is 1, no matter how large the container is.
- MC=1 and SC$\in$[0, 1] $\Rightarrow$ LC=SC. If the memory contribution of a container is extremely high (close to 1), the confidence of this container is decided by its staleness contribution.

Our study shows that this definition of confidence effectively separates containers that are the sources of leaks from those that do not leak. A sample report that includes LC, MC, and SC values for several containers is shown in Table 1. This table is a part of the report generated by our tool when analyzing Sun's bug #6209673. The first container in the table is the one that actually leaks memory. Note that the LC value of this container is much larger than the LC values for the remaining containers. Using this report, it is straightforward to find and fix this bug.

```
class HashMap {
  Object put(Object key, Object value) {...}
  Object get(Object key) {...}
  Object remove(Object key) {...}
  ... }
            (a) Container class HashMap

class Java_util_HashMap {
  static void put_after(int csID, Map receiver, Object key,
                        Object value, Object result) {
     /* if key does not exist in the map */
     if (result == null) {
       /* use user-defined hash code as ID */
       Recorder.v().useUserDefHashCode();
       /* record operation ADD(receiver,key) */
       Recorder.v().record(csID, receiver, key,
                    receiver.size()-1, Recorder.EFFECT_ADD);
     }}
  static void get_after(int csID, Map receiver, Object key,
                        Object result) {
     /* if an entry is found */
     if (result != null) {
       Recorder.v().useUserDefHashCode();
       /* record operation GET(receiver):key */
       Recorder.v().record(csID, receiver, key, receiver.size(),
                           Recorder.EFFECT_GET);
     }}
  static void remove_after(int csID, Map receiver, Object key,
                           Object result) {
     if (result != null) {
       Recorder.v().useUserDefHashCode();
       /* record operation REMOVE(receiver,key) */
       Recorder.v().record(csID, receiver, key,
                    receiver.size()+1, Recorder.EFFECT_REMOVE);
     }}}
              (b) Glue class for HashMap
```

Figure 3: Modeling of container `java.util.HashMap`.

3. MEMORY LEAK DETECTION FOR JAVA

Based on the leak confidence analysis, this section presents a memory leak detection technique for Java.

Container modeling. We have built the glue code for all types in the Java collections framework. For each container type there is a corresponding "glue" class. For each method in the container type that is related to ADD, GET, and REMOVE operations, there is a static method in the glue class whose name is the name of the container method plus the suffix "_before" or "_after". The suffix indicates whether calls to the glue method should be inserted before or after call sites invoking the original method. The parameter list of the glue method includes a call site ID, the receiver object, and the formal parameters of the container method. For the suffix "_after", the return value of the container method is also added. Figure 3 shows the modeling of container class `java.util.HashMap`. It is important to note that most of this glue code can be generated automatically using predefined code templates.

The glue methods call our profiling library to pass the following data: the call site ID (`csID`), the container object, the element object, the number of elements in the container before the operation is performed, and the operation type. The call site ID is generated by our tool during instrumentation. The container object, the element object, and the operation type are used to compute the SC for the container. Recording the number of elements in a container is needed for the algorithm in Figure 5, as discussed later. We use an integer ID to track each object (i.e., container and element). The first time a container object is observed by the profiling library, we tag this object with the ID using JVMTI. The ID for a container object (e.g., the object referred to by

Name	*Description*	*Purpose*
GC_T	GC timestamps	To identify the leaking region
GC_M	Total live memory after GCs	To identify the leaking region
CON_M	Memory taken up by containers	To compute MC for containers
CON_T	Timestamps when measuring CON_M	To compute MC for containers
CON_A	Allocation times of containers	To compute MC and SC for containers
CON_D	Deallocation times of containers	To compute MC and SC for containers
OPR	Operations (csID, container, element, #elements, type)	To compute SC for containers

Table 2: Data collected by the profiler.

`receiver` in Figure 3) is its identity hash code determined by its internal address in the JVM. For an element object, the identity hash code is used as element ID if the container does not have hash-based functions; otherwise, the element ID is the user-defined hash code. For example, in Figure 3, calls to `useUserDefHashCode` inform our library that the ID for `key` should be its user-defined hashcode. For `HashMap`, we only track `key` as a container element, because `key` is representative of a map entry. Methods that retrieve the entire set of elements, such as `toArray` and `iterator`, are treated as a set of GET operations performed on all container elements. Of course, this approximate treatment of iterators may affect the precision of the computed SC values.

Instrumentation. The Soot analysis framework [24] is used to perform code instrumentation. For each call site in an application class at which the receiver is a container, calls to the corresponding glue method are inserted before and/or after the site. For a container object, code is also inserted after its allocation site in order to track its allocation time.

Naively instrumenting a Java program can cause tracking of many containers, which may introduce significant run-time overhead. Because thread-local and method-local containers[1] are less likely to be sources of leaks, we employ an escape analysis to identify a set S of thread-local and method-local objects. We do not instrument a call site if the points-to sets of its receiver variable is a subset of S.

Profiling. Table 2 lists the types of data that need to be obtained by our profiler. In order to identify the leaking region, we need to collect GC finishing times (GC_T) and live memory at these times (GC_M), using JVMTI.

For MC values of containers, it is necessary to collect the amounts of memory for the entire object graphs of containers (CON_M) and the corresponding collection times (CON_T). We measure the memory usage of a container by traversing the object graph starting from the container, using reflection. Since it is impractical to compute the exact value of MC, sampling is used to approximate the memory time graph. Frequent sampling results in precise approximation, but increases run-time overhead.

We launch periodic object graph traversals (for a set of tracked containers) every time after a certain number of gc-events is seen. The number of gc-events between two traversals (i.e., the sampling rate) can be given as a parameter to determine precision and overhead. Our study indicates that choosing 50 as the number of gc-events between traversals can keep the overhead low while achieving high precision.

Object graph traversal is performed by a separate thread. Once a container operation occurs (i.e., `record` in Figure 3 is invoked), `record` adds the ID of the container to a global queue, if that ID is not already there. When the given number of gc-events complete, the JVMTI agent activates this thread, which in turn suspends all other threads, reads IDs from the queue, retrieves the corresponding objects, and performs graph traversals. The allocation time of a container (CON_A) can be collected by the instrumentation at the allocation site, and the JVMTI agent can provide the deallocation time of a tagged container (CON_D).

[1]Containers that are not reachable from multiple threads, and whose lifetime is limited within their allocating methods.

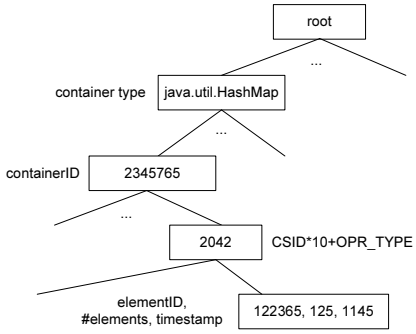

Figure 4: Compressed recording of OPR events.

To compute SC values for containers, we have to record every operation that a tracked container performs (OPR). Because OPR events can result in large amounts of data, we use a data compression strategy to reduce space overhead. The OPR data is stored in a tree structure. Data at a higher level of the tree is likely to be more frequently repeated. For example, type `java.util.HashMap`, which is at the highest level of the tree, appears in the event sequence for many container IDs. Similarly, for a single container ID, many call sites and operations need to be recorded. The tree representation is illustrated in Figure 4. The type of container is a parent of the container ID. A child of the container ID is a combination of the call site ID and the operation type (encoded as a single integer `csID*10+opr_type`). The leaf nodes contain tuples of element ID, number of elements in the container before this operation, and a timestamp.

Keeping too much profiling data in memory degrades program performance. We periodically dump the data to disk to reduce its influence on the run-time execution. The frequency of dumping is the same as that of object graph traversal: the JVMTI agent creates a dumping thread that is activated at the same time the graph traversal thread is activated. Both of these threads must complete before the execution of the application threads is resumed.

Data analysis. Our implementation performs an offline analysis after the program finishes or runs out of memory. Thus, the end of the leaking region τ_e is the ending time of the program. The implementation can easily be adapted to run the analysis online (in another process) and generate the report while the original program is still running.

The first step of the analysis is to scan GC_T and GC_M information to determine the leaking region. The current

```
1: FIND_SC(Double τe, Double τs, Map size_map, Map oper_map)
2: /* operation list for each container */
3: List oper_list
4: /* The result map contains each container ID and its SC */
5: Map result = ∅
6: for each container ID c in oper_map do
7:   Map temp = ∅ /* a temporary helper map */
8:   oper_list = oper_map.get(c)
9:   Integer total = 0 /* total number of elements */
10:  Double sum = 0 /* ∑ staleness */
11:  /* Number of elements in c at time τs */
12:  Integer ne = size_map.get(c)
13:  for each operation opr in oper_list do
14:    if opr.type == "ADD" then
15:      temp.add(opr.elementID, opr.timestamp)
16:    end if
17:    if opr.type == "GET" then
18:      update temp with (opr.elementID, opr.timestamp)
19:    end if
20:    if opr.type == "REMOVE" then
21:      if temp.contains(opr.elementID) then
22:        Integer lastget = temp.get(opr.elementID)
23:        sum += opr.timestamp − lastget
24:        total += 1
25:        temp.remove(opr.elementID)
26:      else
27:        /* The element is added before τs */
28:        sum += opr.timestamp − τs
29:        total += 1
30:        ne −= 1
31:      end if
32:    end if
33:  end for
34:  if temp.size > 0 then
35:    /* These elements are never removed */
36:    for each elementID in temp do
37:      Integer lastget = temp.get(elementID)
38:      sum += τe − lastget
39:      total +=1
40:    end for
41:  end if
42:  if ne > 0 then
43:    /* Elements are added before τs and never removed */
44:    sum += (τe − τs) × ne;
45:    total += ne
46:  end if
47:  c.SC = (sum/total)/(τe − τs)
48:  result.add(c, c.SC)
49: end for
50: return result
```

Figure 5: Computing SC for containers.

implementation employs 0.5 as the ratio used to define this region, which means that at least half of the gc-events form a subsequence with increasing memory consumption (recall the leak region definition from Section 2). After the smallest τ_s that satisfies this constraint is found, each container's OPR data is uncompressed into individual operations and they are sorted by timestamp. The container ID and its operation list are stored in map *oper_map*. For each container, the analysis also determines the first operation that is performed after τ_s; the container ID and the number of container elements at this first operation are stored in map *size_map*. Operations that occured before τ_s are discarded.

For each container, CON_M and CON_T data is used to approximate the memory time graph and the MC value. The approximation assumes that the memory used by the container does not change between two samples. Thus, MC is $\sum_{i=0}^{n-1}(\text{CON}_{T,i+1} - \text{CON}_{T,i}) \times \text{CON}_{M,i}$ where i represents the i-th sample.

Figure 5 shows the computation of SC for containers. The algorithm scans a container's operation list, and for each element ID, finds its last GET operation, its REMOVE operation, and the distance between them. (Recall that the deallocation of the container is treated as a set of REMOVE operations on all elements.) For an element that is added before τ_s (lines 27–30), staleness is the distance between the REMOVE operation and τ_s. For an element that is never removed (lines 34–39), staleness is the distance between τ_e and the last GET operation. For elements that are added before τ_s and never removed (lines 42–45), staleness is $\tau_e - \tau_s$.

Leaking call sites. For each element in a container, the analysis finds the call site ID corresponding to its last GET or ADD operation. Then, it computes the average staleness of elements whose last GET or ADD operations correspond to that same call site ID. The call site IDs are then sorted in decreasing order of this average value. Thus, the tool reports not only the potentially leaking containers (sorted by the LC value), but also, for each container, the potentially leaking call sites (with their source code location) sorted in descending order by their average staleness. Our experience indicates that this information can be very helpful to a programmer trying to identify the source of a memory leak bug.

4. EMPIRICAL EVALUATION

To evaluate the proposed technique for container-based memory leak detection for Java, we performed a variety of experimental studies focusing on leak identification and execution overhead. Section 4.1 illustrates the ability of the technique to help a programmer find and fix real-world bugs. Section 4.2 presents a study of the incurred overhead.

4.1 Detection of Real-World Memory Leaks

The experiments were performed on a 2.4GHz dual-core PC with 2GB RAM. Three different sampling/dumping rates were used: 1/15gc, 1/50gc, and 1/85gc (i.e., once every 15, 50, or 85 gc-events). The experimental subjects were two memory leak bugs reported in the Sun bug database [23], as well as a known leak in SPECjbb [22].

Java AWT/Swing bugs. About half of the memory leak bugs in the JDK come from AWT and Swing. This is the reason we chose two AWT/Swing related leak bugs #6209673 and #6559589 for evaluation. The first bug has already been fixed in Java 6, while the second one was still open and unresolved.

Bug report **#6209673** describes a bug that manifests itself when switching between a running Swing application that shows a JFrame and another process that uses a different display mode (e.g., a screen saver) — the Swing application eventually runs out of memory. According to a developer's experience [18], the bug was very difficult to track down before it was fixed. We instrumented the entire `awt` and `swing` packages, and the test case provided in the bug report. We then ran the instrumented program and reproduced the bug. Figure 6 shows the tool reports with three sampling rates. Each report contains the top three containers, for each container the top three potentially leaking call sites (`---cs`), and the time used to analyze the data.

Sampling rates 1/15gc and 1/50gc produce the same containers, in the same order. The first container in the reports is a `HashMap` in class `javax.swing.RepaintManager`. We inspected the code of `RepaintManager` and found that the container was an instance field `volatileMap`. The call site in the report (with average staleness 0.507) directed us to line 591 in the code of the class, which corresponds to a GET operation `image = (VolatileImage)volatileMap.get(config)`. The tool report indicates that the image obtained at this call

```
Container:11324773 type: java.util.HashMap
        (LC: 0.449, SC: 0.495, MC: 0.825)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.507)
Container:18429817 type: java.util.LinkedList
        (LC: 0.165, SC: 0.194, MC: 0.820)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.246)
Container:8984226 type: java.util.LinkedList
        (LC: 0.051, SC: 0.062, MC: 0.809)
---cs: java.awt.DefaultKeyboardFocusManager:851 (0.063)
---cs: java.awt.DefaultKeyboardFocusManager:740 (0.025)
Data analyzed in 149203ms
              (a) 1/15gc sampling rate

Container:29781703 type: java.util.HashMap
        (LC: 0.443, SC: 0.480, MC: 0.855)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.480)
Container:2263554 type: class java.util.LinkedList
        (LC: 0.145, SC:0.172, MC: 0.814)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.017)
Container:399262 type: class javax.swing.JPanel
        (LC: 0.038, SC:0.044, MC: 0.860)
---cs: javax.swing.JComponent:796 (0.044)
Data analyzed in 21593ms
              (b) 1/50gc sampling rate

Container:15255515 type: java.util.HashMap
        (LC: 0.384, SC:0.426, MC: 0.835)
---cs: javax.swing.RepaintManager:591 (0.426)
Container:19275647 type: java.util.LinkedList
        (LC: 0.064, SC:0.199, MC: 0.244)
---cs: java.awt.SequencedEvent:176 (0.204)
---cs: java.awt.SequencedEvent:179 (0.010)
---cs: java.awt.SequencedEvent:128 (1.660E-4)
Container:28774302 type: javax.swing.JPanel
        (LC: 0.036, SC:0.042, MC: 0.839)
---cs: javax.swing.JComponent:796 (0.042)
Data analyzed in 10547ms
             (c) 1/85gc sampling rate
```

Figure 6: Reports for JDK bug #6209673.

```
Container:5678233 type: java.util.Vector
       (LC: 0.890, SC: 0.938, MC: 0.427)
---cs: java.awt.Window:1825 (0.938)
Container:3841106 type: java.beans.PropertyChangeSupport
       (LC: 0.645, SC:0.779, MC: 0.427)
---cs: java.awt.Component:7007 (0.779)
Container:24333128 type: javax.swing.UIDefaults
       (LC: 0.644, SC:0.875, MC: 0.087)
---cs: javax.swing.UIDefaults:334 (0.868)
---cs: javax.swing.UIDefaults:308 (0.660)
Data analyzed in 454ms
              (a) 1/15gc sampling rate

Container:5678233 type: java.util.Vector
       (LC: 0.890, SC:0.938, MC: 0.427)
---cs: java.awt.Window:1825 (0.938)
Container:30318493 type: java.beans.PropertyChangeSupport
       (LC: 0.668, SC:0.828, MC: 0.288)
---cs: java.awt.Component:7007 (0.828)
Container:9814147 type: javax.swing.UIDefaults
       (LC: 0.101, SC: 0.327, MC: 0.175)
---cs: javax.swing.UIDefaults:334 (0.984)
---cs: javax.swing.UIDefaults:308 (0.903)
Data analyzed in 282ms
              (b) 1/50gc sampling rate

Container:5678233 type: java.util.Vector
       (LC: 0.293, SC:0.425, MC: 0.525)
---cs: java.awt.Window:1825 (0.425)
Container:30502607 type: javax.swing.JLayeredPane
       (LC: 0.117, SC:0.221, MC: 0.441)
---cs: javax.swing.JComponent:796 (0.162)
Container:2665317 type: javax.swing.UIDefaults
       (LC: 0.096, SC:0.363, MC: 0.124)
---cs: javax.swing.UIDefaults:334 (0.359)
---cs: javax.swing.UIDefaults:308 (0.340)
Data analyzed in 297ms
              (c) 1/85gc sampling rate
```

Figure 7: Reports for JDK bug #6559589.

site may not be properly removed from the container. For a programmer that is familiar with the code, this information may be enough to identify the bug quickly.

Since the code was new to us, we had to learn more about this class and the overall display handling strategy of Swing to understand the bug. Because the bug was already resolved, we examined the bug evaluation, which confirmed that `volatileMap` is the root of the leak. The cause of the bug is caching by `RepaintManager` of all VolatileImage objects, regardless of whether or not they are currently valid. Upon a display mode switch, the old GraphicsConfiguration objects under the previous display mode get invalidated and will not be used again. However, the VolatileImage for an obsolete GraphicsConfiguration is never removed from `volatileMap`, and hence all resources allocated by the image continue taking up memory.

Note that the report with sampling rate 1/85gc "loses" the LinkedList in `DefaultKeyboardFocusManager`, which appears as the second container in the other two reports. Although this container is not the source of the bug, it demonstrates that sampling at 1/85gc may not be frequent enough to maintain high precision for LC computation. Also note that analysis time decreases with the decrease in sampling rate, because the tool processes less data.

Compared to our reports, existing approaches that keep track of arbitrary objects (i.e., do not have our container-centric view) would report allocation sites of some types of objects that either (1) continuously grow in numbers or (2) are not used for a while. For bug #6209673, for example, there are growing numbers of objects of numerous types that are reachable by VolatileImage and GraphicsConfiguration objects. Tools such as Cork [14] have to backward-traverse the object graph from the growing objects to find the type of objects that do not grow in numbers. However, the useless objects are inter-referenced, and moreover, traversing back from these growing objects can potentially find multiple types whose instances remain unchanged. In this case, the container that holds GraphicsConfigurations, the JFrame window, the GraphicsDevice object, the map that holds VolatileImages, etc. can all be data structures that are backward-reachable from the growing objects and whose numbers of instances do not grow. Tools such as Sleigh [1] report errors based solely on the staleness of objects. In this case, the JFrame object would be the most stale object because it is never used after it is created. In addition, there are numerous types of objects that are more stale than VolatileImages, such as all components in the frame. Hence, Sleigh could report all these objects as the sources of the leak, including many false positives. Finally, both of these existing approaches require non-standard JVM modifications and support, while our technique uses only code instrumentation and the standard JVMTI interface.

Report **#6559589** describes a bug in Java 6 build 1.6.0_01: calling `JScrollPane.updateUI()` in a Swing program that uses `JScrollPane` causes the number of listeners to grow. Because it is common knowledge that PropertyChangeListeners are managed by `java.bean.PropertyChangeSupport`, we modeled this class as a container and wrote a glue class for it. The reports are shown in Figure 7. The first container in all three reports is a vector in `java.awt.Window`, corresponding to an instance field `ownedWindowList`; this field is used to hold all children windows of the current window. Line 1825 of class `Window` contains an ADD operation `addElement(weakWindow)` for this field. The reporting

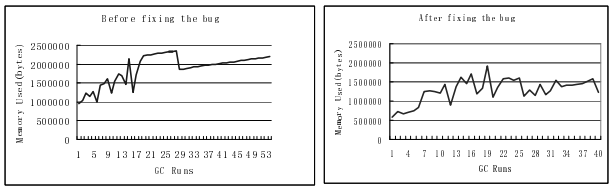

Figure 8: Memory footprint before and after fixing JDK bug #6559589.

of this call site by the tool indicates that when a `Window` object is added to the vector, it may not be properly removed later. We quickly concluded that this cannot be the source of the bug, because windows in a Swing program usually hold references to each other until the program finishes. This forced us to look at the second container in reports (a) and (b), which is a `PropertyChangeSupport` object in `java.awt.Component`. The reported call site at line 7007 of `Component` is

```
changeSupport.addPropertyChangeListener(listener)
```

The container is an instance field `changeSupport`, which stores all PropertyChangeListeners registered in this component. The call site indicates that the bug may be caused by some problem in `JScrollPane` that does not appropriately remove listeners. Registering and unregistering of listeners for `JScrollPane` is done in a set of `ScrollPaneUI` classes. The test case uses a metal look and feel, which is represented by class `MetalScrollPaneUI`, a subclass of `BasicScrollPaneUI`. We checked method `uninstallListeners` in `MetalScrollPaneUI`, which is supposed to release listeners from the component, and found that this method calls the method with the same name in its super class, but does not remove the `scrollBarSwapListener` object held by a private field in the subclass. Further investigation revealed an even more serious problem: method `uninstallListeners` in the subclass was not executed at all, because its signature was different from the signature of the method with the same name in superclass `BasicScrollPaneUI`:

```
/* BasicScrollPaneUI */
void uninstallListeners(JComponent c)
/* MetalScrollPaneUI */
void uninstallListeners(JScrollPane scrollPane)
```

Hence, the causes of the bug are (1) `uninstallListeners` in `MetalScrollPaneUI` fails to override the appropriate method in superclass `BasicScrollPaneUI`, and (2) the listener defined in subclass `MetalScrollPaneUI` is not removed by its own `uninstallListeners`. We modified the code accordingly, and the memory leak disappeared. The memory footprints before and after fixing the bug are shown in Figure 8. We have submitted our modification as a comment in the bug database. Again, the report that used 1/85gc sampling rate failed to include the `PropertyChangeSupport` object, which is the source of the leak.

SPECjbb bug. Benchmark SPECjbb2000 simulates an order processing system and is intended for evaluating server-side Java performance [22]. The program contains a known memory leak bug that manifests itself when running for a long time without changing warehouses. The report generated by our tool for rate 1/50gc is shown in Figure 9. Due to the imprecision of using sampling rate 1/85gc, the report for it is not shown. We also do not show the report for sampling rate 1/15gc, because the containers and their order are the same as in the report for 1/50gc.

```
Container:4451472 type: java.util.Hashtable
        (LC: 0.135, SC: 0.190, MC: 0.659)
---cs: spec.jbb.StockLevelTransaction:225 (0.214)
---cs: spec.jbb.StockLevelTransaction:211 (0.190)
Container:7776424 type: java.util.Hashtable
        (LC: 0.110, SC:0.157, MC: 0.659)
---cs: spec.jbb.StockLevelTransaction:211 (0.157)
---cs: spec.jbb.StockLevelTransaction:225 (0.114)
Container:28739781 type: java.util.Hashtable
        (LC: 0.102, SC:0.146, MC: 0.654)
---cs: spec.jbb.StockLevelTransaction:211 (0.146)
---cs: spec.jbb.StockLevelTransaction:225 (0.122)
Data analyzed in 4078ms
       (a) before modeling of longBTree, using 1/50gc

Container:27419736 type: spec.jbb.infra.Collections.longBTree
        (LC: 0.687, SC: 0.758, MC: 0.666)
---cs: spec.jbb.District:264 (0.826)
---cs: spec.jbb.StockLevelTransaction:225 (0.624)
---cs: spec.jbb.StockLevelTransaction:211 (0.519)
Container:21689791 type: spec.jbb.infra.Collections.longBTree
        (LC: 0.685, SC: 0.757, MC: 0.662)
---cs: spec.jbb.District:264 (0.783)
---cs: spec.jbb.StockLevelTransaction:211 (0.370)
---cs: spec.jbb.District:406 (2.944E-4)
Container:27521273 type: spec.jbb.infra.Collections.longBTree
        (LC: 0.667, SC: 0.727, MC: 0.727)
---cs: spec.jbb.Warehouse:456 (0.798)
---cs: spec.jbb.District:264 (0.784)
---cs: spec.jbb.StockLevelTransaction:211 (0.484)
Data analyzed in 7579ms
       (b) after modeling of longBTree, using 1/50gc
```

Figure 9: Report for SPECjbb2000 bug.

antlr	4.1E-5	chart	2.7E-6	fop	1.3E-5
hsqldb	4.4E-7	jython	5.0E-8	luindex	9.1E-5
lusearch	2.3E-2	pmd	4.3E-6	xalan	5.2E-5
jflex	1.8E-7				

Table 3: Confidences for leak-free programs.

The program was first instrumented without modeling any user-defined containers. The result is shown in Figure 9(a). None of the containers in the report are likely to leak memory, because their confidences are very small. The first container refers to a hashtable that holds stocks of an order line. We did not find any problem with the use of this container. However, we observed that the order lines are actually obtained from an order table, which has a type of `longBTree`. We found that `longBTree` is a container class that implements a BTree data structure and is used to hold orders. It took several minutes to write a glue class for `longBTree`. The program was then re-instrumented and re-executed. The resulting report is shown in Figure 9(b). The top three containers in the report are now instances of `longBTree`.

Line 264 of `spec.jbb.District` is an ADD operation

```
orderTable.put(anOrder.getId(), anOrder)
```

which indicates that `orderTable` may leak memory. Methods `removeOldestOrder`, `removeOldOrders`, and `destroy` contain REMOVE operations for `orderTable`. We focused on the first two methods, because `destroy` could not be called when a district is still useful. Using a standard IDE, we found the callers of these methods: `removeOldestOrder` is called only once within `DeliveryTransaction`, and `removeOldOrders` is never called. Therefore, when a transaction completes, it removes only the oldest order from the table, which causes the heap growth. Inserting code to remove orders from the table fixed the bug. We used less time (a few hours) than the authors of [14] did (a day) to locate the bug in this program, which we had never studied before.

Program	(a)			(b)	(c) 1/15gc				(d) 1/50gc				(e)
	$\#IS$	$\#IS_e$	IT(s)	RT_o(s)	$\#GC_d$	RT_d(s)	$\#GC_l$	RT_l(s)	$\#GC_d$	RT_s(s)	$\#GC_l$	RT_l(s)	$\%OH$
antlr	176	123	87	17.9	387	18.4	10	18.1	387	18.4	10	18.1	0.7%
chart	894	867	202	8.5	5368	38.0	185	35.4	4109	36.5	185	35.1	313.2%
fop	1378	1375	125	4.5	693	8.6	24	7.8	545	8.9	24	6.4	44.6%
hsqldb	684	674	116	4.3	54	4.7	8	4.4	54	4.7	8	4.4	1.6%
jython	443	416	135	7.3	1653	31.8	126	28.2	1440	31.4	126	28.5	298.3%
luindex	442	409	65	19.5	1446	24.4	40	23.7	1390	23.9	40	23.7	21.2%
lusearch	442	388	81	2.9	418	9.1	21	3.9	326	8.2	23	3.2	11.7%
pmd	814	690	111	5.9	2938	26.9	716	18.4	2766	25.2	37	6.6	10.8%
xalan	755	752	114	1.4	655	7.7	30	4.0	605	6.2	18	3.7	165.9%
jflex	522	438	92	45.1	4171	170.7	1493	130.3	2126	165.8	665	88.05	95.2%
bug 1	3109	2768	487	–	18630	600	7420	600	11457	600	1983	600	–
bug 2	3105	2770	502	38.1	512	53.0	243	42.3	413	52.2	37	42	10.5%
specjbb	74	73	142	–	18605	3600	15080	3600	16789	3600	10810	3600	–

Table 4: Overhead: (a) code instrumentation; (b) original running time; (c) running with 1/15gc rate; (d) running with 1/50gc rate; (e) run-time overhead.

Leak-free programs. The tool was also used to analyze several programs that have been used widely and tested extensively for years, and do not have any known memory leaks. Table 3 shows the confidence values computed for these programs. The goal of this experiment was to determine whether the proposed technique produced any false positives for these (almost certainly) leak-free programs. The low confidence values reported by the tool are the expected and desirable outcome for this experiment.

4.2 Static and Dynamic Overhead

This section describes a study of the overhead introduced by the technique. This study utilizes the three bugs described earlier, as well as the 10 programs from Table 3. The instrumented programs were analyzed with rates 1/15gc and 1/50gc. The maximum JVM heap size for each run was set to 512MB (JVM option `Xmx512m`). For each sampling rate, we ran the programs once with the default initial heap size and once with a large initial heap size (JVM option `Xms512m`), in order to observe different numbers of gc-events. The tool reports shown earlier were obtained with the default initial heap size; with the large initial size, the top containers and call sites and their ordering were the same.

Table 4 describes the static and dynamic overhead of the tool. Columns IS and IS_e show the numbers of call sites instrumented without and with employing escape analysis, respectively. Column IT ("instrumentation time") represents the static overhead of the tool — that is, the time (in seconds) it takes to produce the escape-analysis-based instrumented version of the original code. Column RT_o ("running time") contains the original running times of the programs.

The dynamic overhead of the approach is described in the remainder of the table. Columns GC_d and GC_l show the numbers of gc-events with the default and with the large initial heap size, respectively. Similarly, RT_d and RT_l show the program running times with these two choices of initial heap size. Column OH represents run-time overhead introduced by our tool when executed with the most optimal configuration, which corresponds to RT_l in columns (d). For `bug 1` and `specjbb`, we ran the test case for 10 minutes and an hour, respectively, because the execution of these two programs does not terminate.

Applying escape analysis reduces the number of call sites that need to be tracked (the reduction varies from 3 to 124 call sites), while still maintaining reasonable instrumentation time. Using the same sampling rate, running a program with a large initial heap size takes less time, because this configuration reduces the number of gc-events, which in turn reduces the numbers of thread synchronizations, disk accesses, and object graph traversals performed by the dynamic analysis. For the same reasons, decreasing the sampling rate reduces the run-time overhead.

On average, the tool introduced 88.5% run-time overhead for the subject programs (without employing escape analysis, the overhead was very slightly higher). Such overhead is acceptable for debugging, but it may be too high for production runs. One possible approach for reducing the overhead is to selectively instrument a program. Based on the manifestation of the bug, developers may have preferences and hints as to where to focus the effort of the tool. The continuous optimization of the tool is part of our future work. For example, the optimization may focus on executing the object graph traversal thread and the data dumping thread in parallel with the application threads. In addition, the retrieval of a container object from its tag though JVMTI also contributes to the execution overhead. Hence, another direction for future work is to re-implement the tool within an existing open-source JVM, such as the Jikes RMV [11], in order to avoid the overhead caused by JVMTI.

5. RELATED WORK

There is a large body of work devoted to the problem of memory leak detection. The discussion below is restricted to approaches that are most closely related to our technique.

Static analysis can find memory errors such as double frees and missing frees for programs written in non-garbage-collected languages. For example, [2] reduces the memory leak analysis to a reachability problem on the program's guarded value flow graph. Saturn [25], taking another perspective, states memory leak detection as a boolean satisfiability problem. Dor et al. [5] propose a shape analysis to prove the absence of memory leaks in several list manipulation functions. Hackett and Rugina [6] use a shape analysis that tracks single heap cells to identify memory leaks. Orlovich and Rugina [19] propose an approach that starts by assuming the presence of errors, and performs a dataflow analysis to disprove their feasibility. Clouseau [9] is a leak detection tool that uses pointer ownership to describe variables responsible for freeing heap cells, and formulates the analysis as an ownership constraint system. Follow-up work [10] proposes a type system to describe the object ownership for containers, and uses type inference to detect constraint violations. Although both this work and our technique fo-

cus on containers, the target of this previous effort are C and C++ program whereas we are interested in a garbage-collected language. The analysis from [10] does not help detect unnecessary references in a Java program. More generally, all static approaches are limited by the lack of general, scalable, and precise reference/heap modeling. Despite a large body of work on such modeling, it remains an open problem for large real-world systems, with many challenges due to analysis scalability, modeling of multi-threaded behavior, dynamic class loading, reflection, etc.

Dynamic analysis [1, 8, 7, 14, 4, 3, 12, 13, 15] has typically been the weapon of choice for detecting memory leaks in real-world software. However, as described in Section 1 and Section 4, existing techniques have a number of deficiencies. The work in [4, 3, 12, 13] enables visualization of heap objects of different types, but does not provide the ability to directly identify the cause of the memory leak. Existing techniques use growing types [14, 17] (i.e., types whose number of instances continues to grow), object staleness [1], or growing containers [15] to identify suspicious data structures that may contribute to a memory leak. However, in general, a memory leak caused by redundant references is due to a complex interplay of memory growing and staleness and possibly other factors. By considering a single metric which combines both factors, our technique could potentially improve the precision of leak identification. In addition, all existing dynamic-analysis-based leak detection approaches except [15] start by considering the leak symptoms (e.g., growing types or stale objects), and then attempt to trace back to the root cause of the leak. As discussed in the description of the JDK bugs from Section 4, the complexity of such bottom-up tracking makes it hard to generate precise analysis reports, and ultimately puts a significant burden on the programmer. In contrast, our approach is designed from a container-centric point of view — it automatically tracks the suspicious behavior in a top-down manner, by monitoring (1) the object graph reachable from a container, and (2) the container-level operations. This higher level of abstraction, compared to traditional low-level tracking of arbitrary objects, simplifies the difficult task of identifying the sources of memory leaks.

6. CONCLUSIONS

This paper presents a novel technique for detecting memory leaks in Java software. Unlike existing dynamic analyses for leak detection, the proposed approach employs a higher-level abstraction, focusing on container objects and their operations, and uses both memory contribution and staleness contribution to decide how significant is a container's leaking behavior. We present an implementation of this technique and experimental studies demonstrating that the proposed tool can produce precise bug reports at a practical cost. These promising initial results indicate that the technique and any future generalizations are worth further investigation. Future work will focus on optimizations to reduce the run-time overhead. Another possible direction may be ways to identify data structures that act as containers (e.g., using the approach from [16]) instead of relying on the programmer, and to automate the mapping of container methods to the ADT operations. Alternative definitions for LC could also be investigated, and more precise handling of iterators may be desirable. More context information about containers and call sites could make the reports more useful.

Acknowledgments. We would like to thank the ICSE reviewers for their valuable comments and suggestions.

7. REFERENCES

[1] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, 2006.
[2] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI*, pages 480–491, 2007.
[3] W. DePauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *USENIX COOTS*, pages 219–234, 1998.
[4] W. DePauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12(14):1431–1454, 2000.
[5] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *SAS*, pages 115–134, 2000.
[6] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.
[7] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter 1992 USENIX Conference*, pages 125–138, 1992.
[8] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, 2004.
[9] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181, 2003.
[10] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In *ICSE*, pages 252–261, 2006.
[11] *Jikes Research Virtual Machine*. `jikesrvm.org`.
[12] *JProbe*. `www.quest.com/jprobe`.
[13] *JProfiler*. `www.ej-technologies.com`.
[14] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.
[15] *LeakHunter*. `www.wilytech.com/solutions/products`.
[16] N. Mitchell. The runtime structure of object ownership. In *ECOOP*, pages 74–98, 2006.
[17] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP*, pages 351–377, 2003.
[18] E. Nicholas. `weblogs.java.net/blog/enicholas/archive/2006/04/leaking_evil.html`.
[19] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *SAS*, pages 405–424, 2006.
[20] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, pages 291–302, 2005.
[21] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in Java. In *CC*, pages 50–66, 2000.
[22] *SPECjbb2000 Documentation*. `www.spec.org`.
[23] *Sun Bug Database*. `bugs.sun.com/bugdatabase`.
[24] 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 *CC*, pages 18–34, 2000.
[25] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *FSE*, pages 115–125, 2005.