Precise Memory Leak Detection for Java Software Using Container Profiling

GUOQING XU, University of California, Irvine
ATANAS ROUNTEV, Ohio State University

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

In this article we introduce a novel *container-based* heap-tracking technique, based on the fact that many memory leaks in Java programs occur due to incorrect uses of containers, leading to containers that keep references to unused data entries. The novelty of the described work is twofold: (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 technique 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, a known bug in SPECjbb 2000, and an example bug from IBM developerWorks, 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*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Memory leaks, container profiling, leaking confidence

ACM Reference Format:
Xu, G. and Rountev, A. 2013. Precise memory leak detection for java software using container profiling. ACM Trans. Softw. Eng. Methodol. 22, 3, Article 17 (July 2013), 28 pages.
DOI: http://dx.doi.org/10.1145/2491509.2491511

1. INTRODUCTION

Large-scale, object-oriented programs commonly suffer runtime bloat that has conspicuous impact on application performance and scalability [Mitchell et al. 2010]. Such bloat takes many different forms [Xu et al. 2010b]. One important example is the existence of unnecessary references to objects that are no longer used. Although managed

This is a revised and extended version of "Precise Memory Leak Detection for Java Software Using Container Profiling" in *Proceedings of the International Conference on Software Engineering (ICSE'08)*.
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.
Authors' Addresses: G. Xu (corresponding author), University of California, Irvine, CA; email: guoqingx@ics.uci.edu; A. Rountev, Ohio State University, OH.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 1049-331X/2013/07-ART17 $15.00
DOI: http://dx.doi.org/10.1145/2491509.2491511

languages such as Java provide much more reliable memory management, such useless objects cannot be reclaimed by the Garbage Collector (GC), leading to wasted memory space and reduced performance. While a single useless object seems to be insignificant, the accumulation of these objects can lead to increased GC effort (to traverse the object graph) and eventually exhaustion of the heap. This problem is referred to as a *memory leak* in Java—even if such objects will never be used, they are reachable in the heap (i.e., they are not lost) and their memory space can never be reclaimed and is wasted throughout the execution.

Java memory leaks 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 (a well-known deficiency of static analyses), 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 [ej-technologies GmbH 2013], JProbe [Quest Software 2013], and LeakHunter [CA Wily Technology 2013] 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 [DePauw et al. 1998; DePauw and Sevitsky 2000; Jump and McKinley 2007, 2010] and fine-grained object tracking [Hastings and Joyce 1992; Hauswirth and Chilimbi 2004; Qin et al. 2005; Bond and McKinley 2006; Novark et al. 2009; Clause and Orso 2010; Xu et al. 2011].

Of existing dynamic techniques, LeakBot [Mitchell and Sevitsky 2003], Cork [Jump and McKinley 2007, 2010], and Sleigh [Bond and McKinley 2006] represent the state-of-the-art. There are two major research challenges in Java memory leak detection. Imprecision can result if these problems are not appropriately handled.

1.1. Challenges

Definition of memory leak symptom. All the dynamic approaches start with observing memory leak symptoms during the execution. What is a good indicator of a memory leak? Both LeakBot and Cork use heap growth as a heuristic, treating the increase of instances of certain types across garbage collection runs as a memory leak symptom. This could result in false positives, because growing instances are not necessarily true leaks and they may be collected later during the execution. Sleigh, on the other hand, uses staleness (time since last use) to find leaks. This approach could lead to imprecision for infrequently accessed objects. 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.

From-symptom-to-cause diagnosis. All 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 runtime 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 Jump and McKinley [2007], it took the authors a significant amount of time to find the sources of leaks after they read the reports generated by Cork.

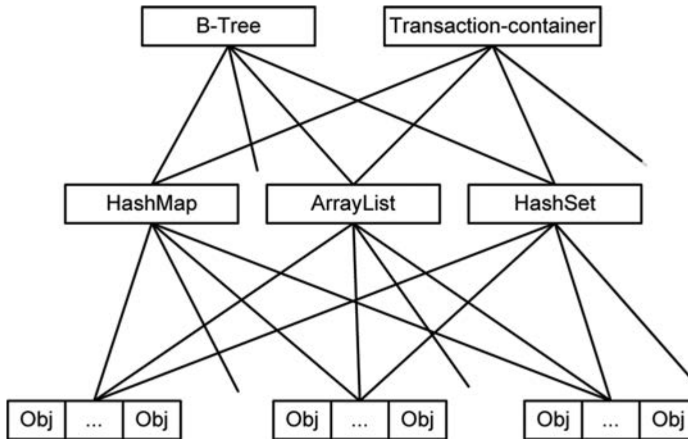

Fig. 1. Container hierarchy in Java.

1.2. Our Proposal

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

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, most of the memory leak bugs reported in the Sun bug repository [Sun bug database 2013] were caused (directly or indirectly) by inappropriate use of containers. We propose a novel technique for Java that detects memory leaks using container profiling. 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 (illustrated in the second 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 are usually sources of leaks. Our technique does not track arrays. Approaches such as Shaham et al. [2000] can be used to complement our technique in order to detect leaks directly caused by arrays.

Our technique requires ahead-of-time lightweight modeling of container behavior: users of the tool 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-supplied glue code to connect

invocations of container methods with our runtime profiling libraries. In order 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 programmers any more than existing memory leak detection techniques: when using existing tools Mitchell and Sevitsky 2003; Bond and McKinley 2006; Jump and McKinley 2007, 2010; Novark et al. 2009; Xu et al. 2011], programmers have to inspect the code to gain similar knowledge about containers so that they can interpret the tool-generated reports. Using our approach simply requires learning such knowledge in advance. Of course, the tool embeds predefined models for containers from the Java collection framework, and therefore programmers need to model only user-defined containers. As shown in our studies, running the tool even without modeling user-defined containers can still provide useful insights for finding leaks: in our reports, top-level Java library containers (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 (the top layer in Figure 1) that are the actual causes of bugs.

Unlike previous approaches, our technique computes a heuristic *leaking confidence* value for each container based on a combination of its memory consumption and the staleness of its data elements, which could yield more accurate results compared to existing approaches [Mitchell and Sevitsky 2003; Bond and McKinley 2006; Jump and McKinley 2007, 2010, Novark et al. 2009; Xu et al. 2011]. For each container, the technique also ranks *call sites* in the source code, based on the average staleness of the elements retrieved at these sites. This container ranking and the related call site ranking provides information that 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 [Sun bug database 2013] and a known memory leak bug in SPECjbb [Standard Performance Evaluation Corporation 2000]—in fact, the top containers in the reports included the ones that leaked memory. In addition, an evaluation of the runtime performance of our technique showed that it has acceptable overhead for practical use.

1.3. Contributions

The main contributions of this work are:

—a dynamic analysis that computes a confidence value for each container, which provides the basis for ranking and reporting of likely-leaking containers;
—a memory leak detection technique for Java based on the confidence analysis;
—a tool that implements the proposed technique;
—an experimental study of leak identification and runtime performance. The results indicate that our technique can precisely detect memory leak bugs with practical runtime overhead.

An earlier version of this work appeared in Xu and Rountev [2008] and Xu [2011].

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. Before describing the details of the analysis, we first provide some basic definitions.

2.1. Definitions

Definition 2.1. (*Container*). A *container type* Γ is an abstract data type with a set Σ of element objects, and three basic operations *ADD*, *GET*, and *REMOVE* that manipulate

Σ. A *container object* γ^n is an instantiation of Γ with n elements in its element set Σ_γ. An element can be of any subtype of O, which denotes the root of the type tree. Both *ADD* and *REMOVE* are mappings of the form $(\Gamma, O) \rightarrow \Gamma$ that map a pair of container object and element object to a container object. *GET* is a mapping $\Gamma \rightarrow O$ from a container object to one of its elements. The effects of the operations are as follows.

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

We treat all (Java library and user-defined) containers as implementations of the container ADT. Here and later in this article, we use the term "container" to denote a container object. Tracking operations on a container requires user-supplied glue classes to bridge the gap between methods defined in the Java implementations and the three basic ADT operations. We have already defined such glue classes 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_i^{gc} and its memory consumption will be denoted by m_i^{gc}. Consider a period of time (referred to as a time region) starting at timestamp τ_s and ending at timestamp τ_e. A program written in a garbage-collected language has a *memory leak symptom* within a time region if GC cannot effectively reduce the memory footprint in this region, and as a result there is a trend of increasing memory consumption. More precisely, a memory leak symptom occurs in time region $[\tau_s, \tau_e]$ if (1) for every gc-event τ_i^{gc} in the region, $m_s \leq m_i^{gc} \leq m_e$, and (2) in this region, there exists a subsequence $ss = (\ldots, \tau_i^{gc}, \ldots, \tau_j^{gc}, \ldots)$ containing n gc-events such that whenever τ_i^{gc} occurs before τ_j^{gc}, we have $m_i^{gc} < m_j^{gc}$. In other words, for the n gc-events included in ss, the memory footprint increases monotonically. The time 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 beginning. Moment of time τ_e can be specified by tool users as an analysis parameter, and can be different for different kinds of analyses. For postmortem offline diagnosis, τ_e is either the ending time of the program or the time when an OutOfMemory error occurs. For online 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 this collected data. We use gc-events as "checkpoints" because at these times the program's memory heap consumption does not include objects that are unreachable.

The definition of a memory leak symptom does not require the amount of consumed memory at each gc-event to be larger than what 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 this user-defined ratio, there could be multiple definitions of starting moment τ_s corresponding to it. Our approach chooses the smallest such value as τ_s, which defines the longest leaking 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. We treat the deallocation of σ^n as being equivalent to n *REMOVE* operations. Thus, a container does not leak memory

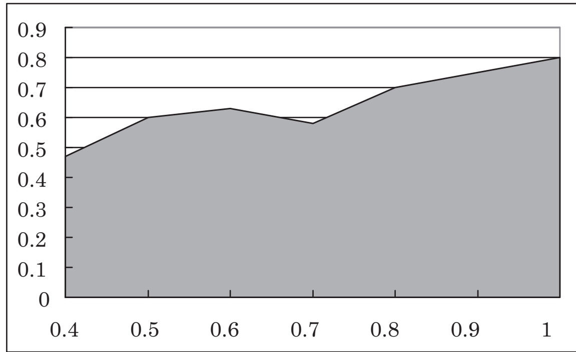

Fig. 2. A sample memory time graph.

if at time τ_e, its accumulated number of *ADD* operations is equal to its accumulated number of *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 preceding 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 much contribution it makes to the memory leak symptom. As mentioned earlier, our confidence value calculation considers both the memory consumption and the staleness when computing the confidence for a container.

2.2. 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 timestamp τ is the ratio between the sum of the memory consumption of all objects that are 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 τ_0 is $max(\tau_s$,allocation time of $\sigma)$, and the ending point of the x-axis is τ_1/τ_e where τ_1 is $min(\tau_e$,deallocation time of $\sigma)$.

A sample graph is illustrated in Figure 2. The x-axis starts at 0.4 relative time (i.e., $0.4 \times \tau_e$ absolute time), which represents either the starting time of the leaking region τ_s, or σ's allocation time, whichever occurs second. The graph indicates that σ does not get freed within the leaking region, because the x-axis ends at 1, which represents the ending time τ_e of this 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 the example in Figure 2, this area is shown in dark. Because the memory time graph starts from τ_s (or later), the MC considers only a container's memory consumption within the leaking region. For a particular 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.

2.3. Staleness Contribution

The second factor that characterizes a container's contribution to the leak is the staleness of the container's elements. The staleness of an object is defined in Bond and McKinley [2006] as the time since the object's last use. Our work provides a new definition of staleness in terms of a container and its elements.

The *staleness* of an element object o in a container σ is $\tau_2 - \tau_1$, where operation REMOVE(σ, o) occurs at τ_2, operation GET(σ):o occurs 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 the 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. Note that the last *GET* operation of a container element may not correspond to the last use site of this object; a reference obtained from the *GET* operation may be stored somewhere else and used later from there. However, it is important to keep in mind that the staleness of the element, as defined before, is a measurement of a container's misbehavior—the object no longer needs to be obtained from the container, but the container still references it. Hence, regardless of whether an object is stored elsewhere via other references, a container behaves properly as long as it removes the object once the container does not need to keep it.

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 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 and not removed before τ_s, relative to the length of the leaking region. Because the staleness of each individual element is at most 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.

2.4. 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 $SC \times MC^{1-SC}$. 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.

Table I. Partial Report of LC, MC, and SC Values

Object ID	*Type*	*LC*	*MC*	*SC*
11324773	java.util.HashMap	0.449	0.824	0.495
18429817	java.util.LinkedList	0.165	0.820	0.194
8984226	java.util.LinkedList	0.050	0.809	0.062
2263554	java.util.WeakHashMap	0.028	0.820	0.034
15378471	java.util.LinkedList	0.018	0.029	0.256
5192610	java.swing.JLayeredPane	0.011	0.824	0.013
30675736	java.swing.JPanel	0.011	0.824	0.013
19526581	java.swing.JRootPane	0.011	0.824	0.013
17933228	java.util.Hashtable	0.000023	0.0007	0.026
33263898	java.util.ArrayList	0.00000026	0.0000032	0.046

—$MC = 0$ 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.
—$SC = 1$ 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.
—$MC = 1$ 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 for several containers is illustrated in Table I. 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. Based on this report, it is straightforward for a programmer to find and fix this bug.

3. MEMORY LEAK DETECTION FOR JAVA

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

3.1. Container Modeling

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.

```
1 class HashMap{
2   Object put(Object key, Object value){...}
3   Object get(Object key){...}
4   Object remove(Object key){...}
5   ...
6 }
```

(a) Container class HashMap

```
7 class Java_util_HashMap{
8   static void put_after(int csID, Map receiver, Object key,
9                         Object value, Object result) {
10      /* if key does not exist in the map */
11      if(result == null){
12        /* use user-defined hash code as ID */
13        Recorder.v().useUserDefHashCode();
14        /* record operation ADD(receiver, key) */
15        Recorder.v().record(csID, receiver, key,
16                     receiver.size()-1, Recorder.EFFECT_ADD);
17      }
18  }
19  static void get_after(int csID, Map receiver, Object key,
20                        Object result){
21      /* if an entry is found */
22      if(result != null){
23        Recorder.v().useUserDefHashCode();
24        /* record operation GET(receiver):key */
25        Recorder.v().record(csID, receiver, key, receiver.size(),
26                            Recorder.EFFECT_GET);
27      }
28  }
29  static void remove_after(int csID, Map receiver, Object key,
30                           Object result){
31      if(result != null){
32        Recorder.v().useUserDefHashCode();
33        /* record operation REMOVE(receiver, key) */
34        Recorder.v().record(csID, receiver, key,
35                     receiver.size()+1, Recorder.EFFECT_REMOVE);
36      }
37  }
38}
```

(b) Glue class for HashMap

Fig. 3. Modeling of container `java.util.HashMap`.

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, the operation type, and the number of elements are used to compute the SC value for the container, as described later. In order to reduce the overhead of runtime bookkeeping, we use an integer ID to track each object (i.e., container and element). The first time a container performs its operation, we tag the container object with the ID (using JVMTI). The ID for a container object (e.g., `receiver` at lines 8, 19, and 29 of Figure 3) is its identity hash code determined by its

Table II. Mapping between Actual Container Methods and Abstract Container Operations

	Container Method Call	*Interpretation*
(a)	A.add(o)	ADD(A, o)
	o=A.get(..)	o=GET(A)
	o=A.remove(..)	REMOVE(A, o)
	A.addAll(B)	$\forall\, o \in B$, o=GET(B) $\forall\, o \in B$, ADD(A, o)
	A.removeAll(B)	$\forall\, o \in B$, o=GET(B) $\forall\, o \in A \cap B$, REMOVE(A, o)
	A.retainAll(B)	$\forall\, o \in B$, o=GET(B) $\forall\, o \in A \setminus B$, REMOVE($A$, o)
	A.containsAll(B)	$\forall\, o \in B$, o=GET(B)
	A.toArray()	$\forall\, o \in A$, o=GET(A)
	A.iterator()	$\forall\, o \in A$, o=GET(A)
(b)	$v = A$.get(k)	k=GET(A) if $v \neq$ null
	$r = A$.put(k, v)	ADD(A, k) if $r =$ null k=GET(A) otherwise
	$r = A$.remove(k)	REMOVE(A, k) if $r \neq$ null
	A.putAll(B)	$\forall\, k \in B.keySet()$, k=GET(B) $\forall\, k \in B.keySet()$: if $k \in A.keySet()$, k=GET(A) otherwise, ADD(A, k)
	A.keySet()	$\forall\, k \in A.keySet()$, k=GET(A)
	A.values()	$\forall\, k \in A.keySet()$, k=GET(A)
	A.entrySet()	$\forall\, k \in A.keySet()$, k=GET(A)
	A.clear()	$\forall\, k \in A.keySet()$, REMOVE($A$, k)

(a) methods defined in `java.util.Collection`; (b) methods defined in `java.util.Map`.

internal address in the JVM. For an element object, the identity hash code is used as its 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` (lines 13, 23, and 32) inform our library that the ID for `key` should be its user-defined hash code. For `HashMap`, we only track `key` as a container element (lines 15, 25, and 34), 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. Note that it may not be precise to model `iterator` in this manner, since a particular object is retrieved only when method `iterator.next()` is actually invoked. Despite its potential imprecision, this treatment avoids the use of heavyweight static (context-sensitive points-to) analysis to relate `Iterator` objects with their corresponding container objects. Methods such as `size` and `isEmpty` are assumed to have no effect on the analysis of leaking behavior, although they depend on the current elements of the container.

The mapping of a typical set of container methods defined in interfaces `java.util.Collection` and `java.util.Map` is illustrated in Table II. Upper-case letters and lower-case letters are used to represent containers and elements, respectively. To construct such a mapping, the method specifications need to be expressed in terms of the basic operations *ADD*, *GET*, and *REMOVE* defined in Section 2.

3.2. Instrumentation

Our tool uses the Soot program analysis framework [Vallée-Rai et al. 2000] to perform code instrumentation. For each call site in an application class at which the receiver

Table III. Data Collected by Our Profiler

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

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 tract the container allocation time.

Naively instrumenting a Java program can cause tracking of a large number of containers, which may introduce significant runtime overhead. Because thread-local and method-local containers[1] are not likely to be the source of a leak, we employ an escape analysis to identify a set S of thread-local and method-local objects. We do not instrument call sites if the points-to sets of their receiver variables are subsets of S. The escape analysis we use is conceptually similar to the one proposed in Choi et al. [1999].

3.3. Profiling

Table III 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). This can be done by using JVMTI agents.

In order to compute MC for containers, we need to collect amounts of memory taken up by 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). As mentioned in Section 2, it is impractical to compute the exact value of MC. Sampling is used during the execution, and the obtained values are used to approximate the memory time graph. Frequent sampling results in precise approximation, but increases runtime 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 can be given as a parameter to our tool to control precision and overhead. Our experimental study indicates that choosing 50 as the number of gc-events between traversals can keep the overhead low while achieving high precision.

Because an object graph traversal can be expensive, this task is assigned to a newly created thread that executes, with appropriate synchronization, in parallel with the main program. Note that such a solution is particularly well-suited for modern

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

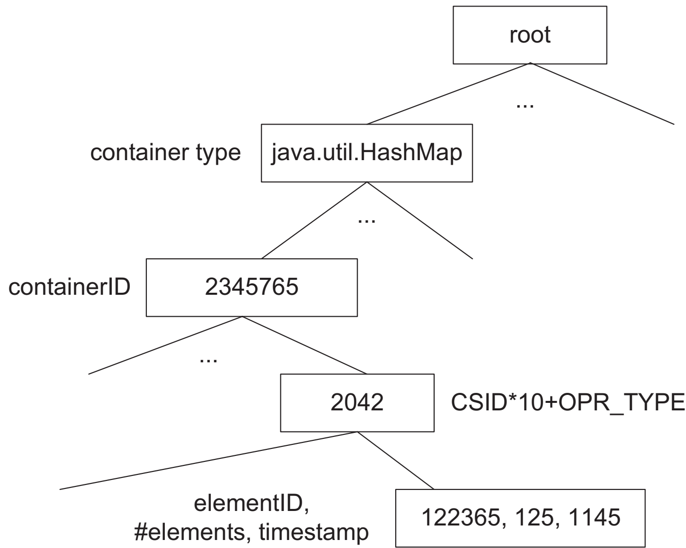

Fig. 4. Compressed recording of OPR events.

architectures with multicore processors. Once a container operation is performed (i.e., `record` in Figure 3 is invoked), `record` adds the ID of the container to a global queue. When the given number of gc-events complete, our JVMTI agent activates this thread, which 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 our JVMTI agent can provide the deallocation time of a tagged container (CON_D).

In order to compute SC 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 record the data to disk to reduce its influence on the runtime execution. The frequency of recording is the same as that of object graph traversal: our JVMTI agent creates a recording thread that is activated at the same time as the graph traversal thread is activated. All the threads synchronize when recording to disk is about to start. As discussed earlier, selecting an appropriate recording (and sampling) rate is key to reducing the runtime overhead.

3.4. Data Analysis

Our current implementation performs an offline analysis of the collected data 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

ALGORITHM 1: Computing SC for containers.

```
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
```

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 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 occurred 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.

Algorithm 1 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 37–39), staleness is the distance between τ_e and the last *GET* operation. For elements that are added before τ_s and never removed (lines 43–45), 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* operation. Then, it computes the average staleness of elements whose last *GET* operations correspond to that same call site ID. These 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 our 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

All experiments were performed on a 2.4GHz dual-core PC with 2GB RAM, running Windows XP and Sun HotSpot JVM version 1.5.0. Three different sampling/recording rates were used: 1/15gc, 1/50gc, and 1/85gc (i.e., once every 15, 50, or 85 gc-events). They were chosen as representatives of their corresponding ranges (0–30, 30–70, and 70–100). The experimental subjects were two memory leak bugs reported in the Sun bug database [Sun bug database 2013], a known leak in SPECjbb [Standard Performance Evaluation Corporation 2000], as well as a bug contained in a leak example from an IBM developerWorks article [Gupta and Palanki 2005].

4.1.1. Java AWT/Swing Bugs. Many 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 was already fixed in Java 6. The second one was still open and unresolved at the time of our evaluation; based on the bug analysis and proposed code changes submitted by us, it was subsequently fixed in Java 7.

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 [Nicholas 2006], 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 5 shows the tool reports with three sampling

```
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

Fig. 5. Reports for JDK bug #6209673.

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 called `volatileMap`. The call site in the report (with average staleness 0.507) directed us to line 591 in the source code of the class, which corresponds to the following *GET* operation.

```
image = (VolatileImage)volatileMap.get(config)
```

The tool report indicates that the image obtained at this call site may not be properly removed from the container. For a programmer who is familiar with the code, this information may be enough to identify the bug quickly. Since the code was new for

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 that all VolatileImage objects are cached by `RepaintManager`, regardless of whether or not they are 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 until an OutOfMemory error occurs.

Note that the report with sampling rate 1/85gc "loses" the LinkedList in class `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. Note that analysis time decreases with the decrease in sampling rate, because the tool loads and processes less data during the analysis.

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 [Jump and McKinley 2007, 2010] 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 [Bond and McKinley 2006] report errors based solely on the staleness of objects. In this case, there are numerous types of objects that are more stale than VolatileImages, such as all the 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 nonstandard JVM modifications and support, while our technique uses only code instrumentation and the standard JVMTI interface.

Currently, the report generated by our tool does not contain calling context information, which could be useful in locating the cause of a bug. This, in fact, does not undermine the practical effectiveness of our technique. Unlike tools that track arbitrary objects and therefore need this information to locate the bug-inducing operations, our tool pinpoints the cause containers and the last *GET* operations, which are strong indications of the location of the bug. For instance, many call chains reported by Sleigh go from the last use sites of stale objects backward to call sites that invoke container methods, which are important to investigate. These call chains may not be as useful if the call sites corresponding to container operations are directly reported (as in our tool).

Report #6559589 describes a bug in Java 6 build 1.6.0_01: calling method `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 generated reports are shown in Figure 6. The first container in all three reports is a vector in `java.awt.Window`, which corresponds to an instance field `ownedWindowList`. Line 1825 of `Window` contains an *ADD* operation.

```
ownedWindowList.addElement(weakWindow)
```

```
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

Fig. 6. Reports for JDK bug #6559589.

Field `ownedWindowList` is used to hold all children windows of the current window. The reporting 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 as follows.

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

The container is an instance field `changeSupport`, which stores all PropertyChange-Listeners 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

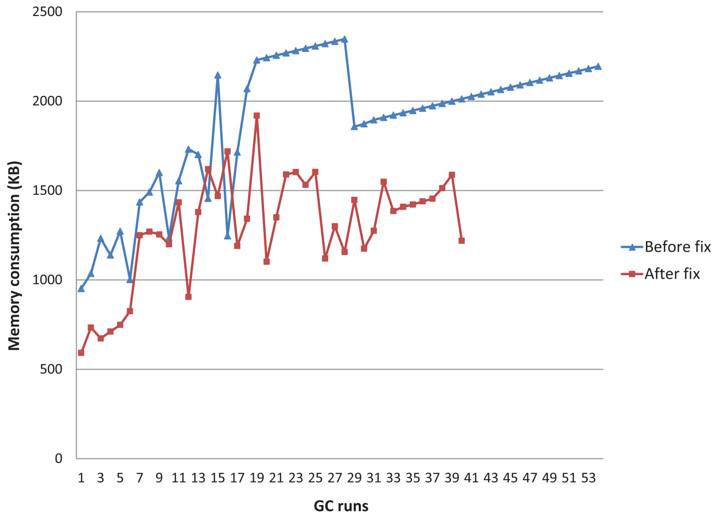

Fig. 7. Memory footprint before and after fixing JDK bug #6559589.

`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 superclass, 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 method `uninstallListeners`. We modified the code accordingly, and the memory leak disappeared. The memory footprint before and after fixing the bug is illustrated in Figure 7. Note that the number of gc-events is smaller for the modified program, since the memory usage is reduced and garbage collection occurs less frequently. We submitted our modification as a comment in the bug database, and the Java 7 release contains our proposed fix for this problem. Again, the report that used 1/85gc sampling rate failed to include the `PropertyChangeSupport` object, which is the source of the leak.

4.1.2. SPECjbb Bug. Benchmark SPECjbb2000 simulates an order processing system and is intended for evaluating server-side Java performance [Standard Performance Evaluation Corporation 2000]. 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 8. Due to the imprecision of using sampling rate 1/85gc, the report for it is not shown. We also do not show the

```
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)
```

(b) after modeling of longBTree, using 1/50gc

Fig. 8. Report for the SPECjbb2000 bug.

report of using sampling rate 1/15gc, because the containers and their order in this report are the same as in the report for 1/50gc.

The program was first instrumented without modeling any user-defined containers. The result is shown in Figure 8(a). It is straightforward to see that none of the containers in the list is likely to leak memory, because their confidence values are very small. The first container in the report refers to a hash table 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 us several minutes to write a glue class for `longBTree`. We then reinstrumented and reran the program. The resulting tool report is shown in Figure 8(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)
```

This 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

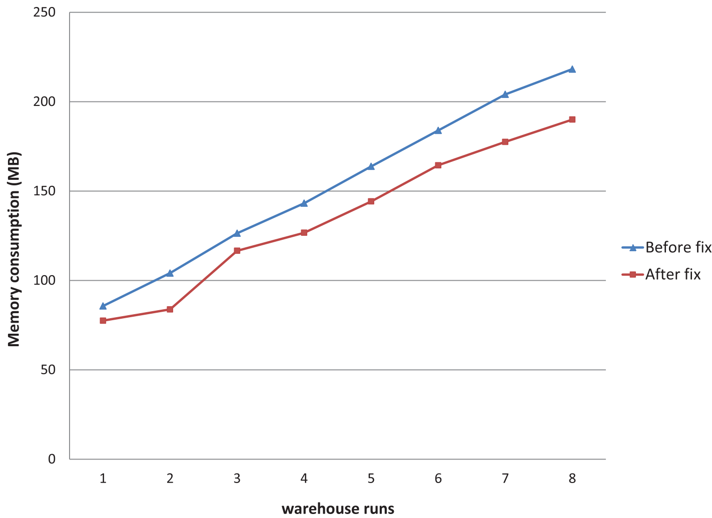

Fig. 9. Memory footprint before and after fixing the SPECJbb memory leak.

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. Inserting code to remove orders from the table fixed the bug. The memory footprint comparison between the original run and the modified run is shown in Figure 9. Since the program continues to load and process warehouse data, the memory footprint increases in both runs, but the rate of this increase is lower in the modified version.

4.1.3. Memory Leak from Java developerWorks. An IBM developerWorks article [Gupta and Palanki 2005] contains a sample leak bug that uses behaviors of three containers to illustrate different levels of leaking severity. Specifically, the first container never removes elements. The second container removes elements after they become useless, but its usage has an error that leads to several Integer objects not being removed in each iteration. The third container removes all elements before the end of each iteration. The program executes in iterations and exercises operations of all the three containers in each iteration. We modified the program by adding small objects (arrays of 1000 integers) to the first container (that causes the quick leak), adding large objects (arrays of 10000 integers) to the second container (that causes the slow leak), and adding even larger objects (arrays of 50000 integers) to the third container. Although this is not a real-world leak bug, the presence of leaks of multiple levels of severity is useful for us to evaluate the sensitivity of the proposed leak confidence model in separating containers that have different contributions to the leak. We instrumented the program and ran it until an OutOfMemory error was caught. The generated report for this program is shown in Figure 10.

Note that the leak confidence computed for the first container is much larger than those computed for the remaining two containers. The primary reason is that this container does not remove any elements during the execution. In other words, its large staleness contribution dominates the fact that the other two containers contain large objects. It is interesting to see that the third (leak-free) container is ranked higher than the second (slowly leaking) one: they have similar staleness contributions, but the third one consumes much more memory. According to this report, the third container is actually the one to which we need to pay more attention, because it has more significant

```
Container:18296328 type: class java.util.ArrayList
        (LC: 0.921, SC: 0.985, MC: 0.011)
---cs: LeakExample:25 (0.327)   // first container

Container:24764524 type: class java.util.ArrayList
        (LC: 0.008, SC:0.035, MC: 0.241)
---cs: LeakExample:57 (0.031)  // third container

Container:16224256 type: class java.util.ArrayList
        (LC: 0.007, SC: 0.037, MC: 0.180)
---cs: LeakExample:39 (0.033)  // second container
Data analyzed in 47275ms

                         (a) 1/15gc sampling rate
```

```
Container:18296328 type: class java.util.ArrayList
        (LC: 0.845, SC: 0.971, MC: 0.01)
---cs: LeakExample:25 (0.327)   // first container

Container:12077888 type: class java.util.ArrayList
        (LC: 0.008, SC:0.031, MC: 0.245)
---cs: LeakExample:57 (0.031)  //third container

Container:16224256 type: class java.util.ArrayList
        (LC: 0.005, SC: 0.033, MC: 0.141)
---cs: LeakExample:39 (0.032)   //second container
Data analyzed in 26198ms

                         (b) 1/50gc sampling rate
```

Fig. 10. Report for the leak bug from IBM developerWorks.

Table IV. Confidences for Leak-Free Programs

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				

influence on runtime performance than the second one. Removing elements earlier, after they become useless (instead of waiting until the end of an iteration) can fix the problem. From this experiment, it can be seen that the leak confidence model may be able to provide an interesting characterization of certain performance issues related to containers.

4.1.4. 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 IV shows the confidence values computed for these programs. The goal of this experiment was to determine whether the tool produced any false positives on these (almost certainly) leak-free programs. The low confidence values reported in the table are the expected and desirable outcome for this experiment.

4.2. Static and Dynamic Overhead

This section describes our study of the overhead introduced by our technique. This study utilizes the three real-world bugs described earlier, as well as a set of Java programs shown in Table IV. The benchmark set includes 9 programs from the DaCapo suite [DaCapo Benchmarks 2006], and jflex, a lexer generator. For each DaCapo program, we instrumented all the application classes and ran it with default workload size. The input for jflex is a grammar file corresponding to a finite state machine with 21769 states. All instances of container classes from package `java.util` were tracked by the analysis.

Table V. Static Analysis Running Time

Program	*#IS*	$\#IS_e$	*IT* (s)
antlr	176	123	87
chart	894	867	202
fop	1378	1375	125
hsqldb	684	674	116
jython	443	416	135
luindex	442	409	65
lusearch	442	388	81
pmd	814	690	111
xalan	755	752	114
jflex	522	438	92
bug 1	3109	2768	487
bug 2	3105	2770	502
specjbb	74	73	142

Table VI. Dynamic Overhead I

Program	(a)	(b) 1/15gc				(c)	
	RT_o (s)	$\#GC_d$	RT_d (s)	$\#GC_l$	RT_l (s)	$\%OH_d$	$\%OH_l$
antlr	17.9	387	18.4	10	18.1	2.8%	1.1%
chart	8.5	5368	38.0	185	35.4	347.0%	316.4%
fop	4.5	693	8.6	24	7.8	91.1%	73.3%
hsqldb	4.3	54	4.7	8	4.4	9.3%	2.3%
jython	7.3	1653	31.8	126	28.2	335.6%	286.3%
luindex	19.5	1446	24.4	40	23.7	25.1%	21.5%
lusearch	2.9	418	9.1	21	3.9	213.8%	34.5%
pmd	5.9	2938	26.9	716	18.4	355.9%	211.9%
xalan	1.4	655	7.7	30	4.0	450%	185.7%
jflex	45.1	4171	170.7	1493	130.3	278.5%	188.9%
bug 1	–	18630	600	7420	600	–	–
bug 2	38.1	512	53.0	243	42.3	39.1%	11.0%
specjbb	–	18605	3600	15080	3600	–	–
average	–	–	–	–	–	195.3%	121.2%

(a) original running time; (b) running with 1/15gc rate; (c) runtime overhead.

We ran the instrumented programs 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 (32Mb) and once with a large initial heap size (JVM option `Xms512m`), in order to observe different numbers of gc-events. We checked the generated reports under the four configurations (i.e., 1/15gc with 512M heap, 1/50gc with 512M heap, 1/15gc with default heap, and 1/50gc with default heap) and found that for all programs, the top five containers reported by our tool under these configurations are the same. The remaining containers (other than the top five) in each report had fairly small confidence values, and therefore, we can assume that the precision loss under configurations with 512M initial heap or 1/50gc sampling rate is small enough so that it is not problematic in the practical use of the tool.

Table V, Table VI, and Table VII describe the static analysis running time, the dynamic overhead of the tool when using 1/15gc, and when using 1/50gc sampling rate, respectively. In Table V columns *IS* and IS_e represent 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

Table VII. Dynamic Overhead II

Program	(a)	(b) 1/50gc				(c)	
	RT_o (s)	$\#GC_d$	RT_d (s)	$\#GC_l$	RT_l (s)	$\%OH_d$	$\%OH_l$
antlr	17.9	387	18.4	10	18.1	2.8%	1.1%
chart	8.5	4109	36.5	185	35.1	329.4%	312.9%
fop	4.5	545	8.9	24	6.4	97.8%	42.2%
hsqldb	4.3	54	4.7	8	4.4	9.3%	2.3%
jython	7.3	1440	31.4	126	28.5	330.1%	290.4%
luindex	19.5	1390	23.9	40	23.7	22.6%	21.5%
lusearch	2.9	326	8.2	23	3.2	182.8%	10.3%
pmd	5.9	2766	25.2	37	6.6	327.1%	11.9%
xalan	1.4	605	6.2	18	3.7	342.9%	164.3%
jflex	45.1	2126	165.8	665	88.05	267.6%	95.2%
bug 1	–	11457	600	1983	600	–	–
bug 2	38.1	413	52.2	37	42	37.0%	10.2%
specjbb	–	16789	3600	10810	3600	–	–
average	–	–	–	–	–	177.2%	87.5%

(a) original running time; (b) running with 1/50gc rate; (c) runtime overhead.

(in seconds) it takes to produce the escape-analysis-based instrumented version of the original code.

Column RT_o ("running time") in both Table VI and Table VII contains the original running times of the programs. The dynamic overhead of the approach is described in the remainder of these two tables. 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. Columns OH_d and OH_l represent the runtime overhead introduced by our tool. Specifically, these two values are computed as follows.

$$OH_d = (RT_d - RT_o)/RT_d$$
$$OH_l = (RT_l - RT_o)/RT_l$$

The average overhead for each configuration is shown at the bottom of the tables. 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. For the purpose of illustration and comparison, the dynamic overhead under different configurations is shown in Figure 11.

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 reason, decreasing the sampling rate reduces the runtime overhead.

By employing both larger initial heap and smaller sampling rate, the average runtime overhead for the programs can be reduced to 87.5%. Such overhead is acceptable for bug detection, but it may be too high for production runs. One possible approach to reducing 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. For example, certain parts of the program that are decided not to be the cause of the bug do not need to be instrumented and tracked at runtime. Additional optimizations of the tool present interesting opportunities for future work. For example, the optimizations may focus on reducing threads synchronizations within the dynamic analysis. Our

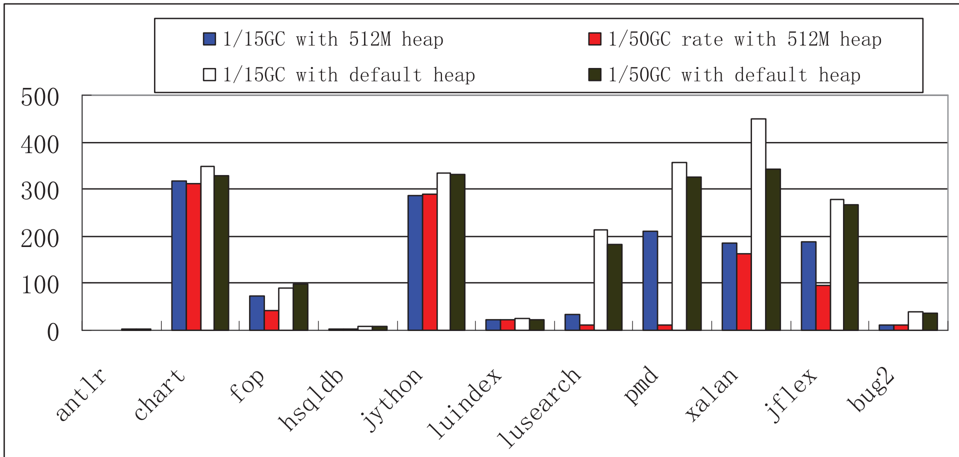

Fig. 11. Dynamic overhead under different configurations.

current implementation uses JVMTI, which runs agents and invokes event callbacks in threads. The running of these threads adds synchronization overhead. In addition, the retrieval of a container object from its tag through JVMTI also contributes to the execution overhead. Hence, a possibility for future work is to reimplement the tool within an existing open-source JVM, such as the Jikes RMV [Jikes Research Virtual Machine 2013], 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 that follows is restricted to approaches that are most closely related to our technique.

5.1. Static Analysis

Static analysis can find memory errors such as double frees and missing frees for programs written in non-garbage-collected languages. For example, Cherem et al. [2007] reduce the memory leak analysis to a reachability problem on the program's guarded value flow graph, and detect leaks by identifying value flows from the source (malloc) to the sink (free). Saturn [Xie and Aiken 2005], taking another perspective, reduces the problem of memory leak detection to a boolean satisfiability problem, and uses an SAT-solver to identify potential bugs. Dor et al. [2000] propose a shape analysis based on 3-valued logic, to prove the absence of memory leaks in several list manipulation functions. Hackett and Rugina [2005] use a shape analysis that tracks single heap cells to identify memory leaks. Orlovich and Rugina [2006] propose an approach that starts by assuming the presence of errors, and performs a backward dataflow analysis to disprove their feasibility. Clouseau [Heine and Lam 2003] 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. Its follow-up work [Heine and Lam 2006] proposes a type system to describe the object ownership for polymorphic containers and uses type inference to detect constraint violations. Although both this work and our technique focus on containers, the targets of this previous effort are C and C++ programs whereas we are interested in a garbage-collected language. The analysis described in Heine and Lam [2006] 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 analysis of large real-world systems, with many challenges due to analysis scalability, modeling of multithreaded behavior, dynamic class loading, reflection, etc.

5.2. Dynamic Analysis

Dynamic analysis (e.g., Hastings and Joyce [1992], DePauw et al. [1998], DePauw and Sevitsky [2000], Hauswirth and Chilimbi [2004], Bond and McKinley [2006], Jump and McKinley [2007, 2010], and Rayside and Mendel [2007]) has typically been the "weapon of choice" for detecting memory leaks in real-world Java software. However, existing techniques have a number of deficiencies. The work in DePauw et al. [1998] and DePauw and Sevitsky [2000] and tools such as Quest Software [2013] and ej-technologies GmbH [2013] enable visualization of objects of different types on the heap, but do not provide the ability to directly identify the cause of the memory leak.

Existing diagnostic techniques for Java use growing types [Mitchell and Sevitsky 2003; Jump and McKinley 2007, 2010] (i.e., types whose number of instances continues to grow) or object staleness [Bond and McKinley 2006] 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 growth, 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 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, the complexity of such bottom-up tracking makes it hard to generate precise analysis reports, and ultimately puts a significant burden on the programmer. Novark et al. [2009] find memory leaks and bloat in C/C++ programs by segregating objects based on their allocation contexts and staleness. Work in Clause and Orso [2010] uses a tainting framework to propagate relevant information in order to identify leaking objects for C/C++ programs. Maxwell et al. [2010] develop a graph mining algorithm that analyzes heap dumps to automatically identify subgraphs which could represent potential memory leak sources.

In contrast, the approach described in this article is designed to account for higher-level semantics: it takes a container-centric view, tracking automatically the suspicious behavior in a top-down manner by monitoring (1) the object graph reachable from a container, and (2) the container-level operations. Other semantics-aware performance analyses (such as Mitchell et al. [2009] and Xiao et al. [2011]) have been developed since the initial publication of our work [Xu and Rountev 2008]. Our recent memory leak detector, LeakChaser [Xu et al. 2011], detects leaks by allowing programmers to explicitly express their interest in certain behaviors (i.e., related to higher-level semantics). This approach is based on the insight that developers' knowledge is essential for a leak detector to produce highly relevant reports. The tool focuses on repeatedly occurring transactions for which object lifetime relationships must be asserted (e.g., that one object dies before another). These higher levels of abstraction—a transaction-centric view in LeakChaser, or a container-centric view in this article—simplify the difficult task of identifying the sources of memory leaks, compared to traditional low-level tracking of arbitrary objects.

5.3. Software Bloat Analysis

As a more general problem [Mitchell et al. 2010; Xu et al. 2010b; Xu 2011], software bloat analysis [Mitchell 2006; Mitchell and Sevitsky 2007; Xu and Rountev 2008, 2010, Xu et al. 2009, 2010a; Mitchell et al. 2009; Shankar et al. 2008; Altman et al. 2010;

Xiao et al. 2011; Yan et al. 2012] attempts to find, remove, and prevent performance problems due to inefficiencies in the code execution and the use of memory. Prior work [Mitchell et al. 2006; Mitchell and Sevitsky 2007] proposes metrics to provide performance assessment of use of data structures. Mitchell et al. [2006] propose a manual approach that detects bloat by structuring behavior according to the flow of information, and their later work [Mitchell and Sevitsky 2007] introduces a way to find data structures that consume excessive amounts of memory. Work by Dufour et al. [2007; 2008] uses a blended escape analysis to characterize and find excessive use of temporary data structures. By approximating object lifetimes, the analysis has been shown useful in classifying the usage of newly created objects in the problematic areas. Shankar et al. propose Jolt [Shankar et al. 2008], an approach that makes aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Work by Xu et al. [2009, 2010a] detects bloat by analyzing chains of copy operations, and computations with imbalanced costs and benefits. Other work [Shacham et al. 2009] dynamically identifies inappropriately used Java collections and recommends to the user those that should be used instead. The technique presented in our work can also be considered as a form of bloat analysis, and it can be used to find a specific type of bloat caused by unnecessary references.

6. CONCLUSIONS AND FUTURE WORK

This article presents a novel technique for detecting memory leaks for Java. Unlike existing “from-symptom-to-cause” dynamic analyses for leak detection, the proposed approach employs a higher-level abstraction, focusing on container objects and their operations. The approach uses both memory usage and staleness to decide whether the leaking behavior of a container is significant. We present an implementation of this technique and a set of experimental studies, demonstrating that the proposed tool can produce precise bug reports at a practical cost. These promising results indicate that the technique and any future generalizations are worth further investigation.

In this work, method semantics is provided by programmers in the form of glue code. We have developed a static analysis [Xu and Rountev 2010] that can automatically infer such semantic information from the program source code. It is worth further investigation on how to employ this automated approach in our leak detection techniques in order to reduce programmers' effort.

This is the first memory leak detection technique that explicitly uses higher-level semantic information to help problem diagnosis. Given the size and complexity of today's large-scale applications, there is little hope that a completely automatic analysis can precisely identify performance bottlenecks. We believe that developers' insight about such higher-level semantics is key to improving the precision and real-world usefulness of similar performance analysis tools.

REFERENCES

ALTMAN, E., ARNOLD, M., FINK, S., AND MITCHELL, N. 2010. Performance analysis of idle programs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*. 739–753.

BOND, M. D. AND MCKINLEY, K. S. 2006. Bell: Bit-encoding online memory leak detection. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 61–72.

CA WILY TECHNOLOGY. 2013. Wily introscope leakhunter. http://www.adhoc-international.com/pages/en/structure/download/cawily_introscope2.pdf.

CHEREM, S., PRINCEHOUSE, L., AND RUGINA, R. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. 480–491.

CHOI, J., GUPTA, M., SERRANO, M., SREEDHAR, V., AND MIDKIFF, S. 1999. Escape analysis for java. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. 1–19.

CLAUSE, J. AND ORSO, A. 2010. Leakpoint: Pinpointing the causes of memory leaks. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*. 515–224.

DACAPO BENCHMARKS. 2006. DaCapo benchmarks. http://www.dacapo-bench.org.

DEPAUW, W., LORENZ, D., VLISSIDES, J., AND WEGMAN, M. 1998. Execution patterns in object-oriented visualization. In *Proceedings of the USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*. 219–234.

DEPAUW, W. AND SEVITSKY, G. 2000. Visualizing reference patterns for solving memory leaks in java. *Concurr. Pract. Exper. 12,* 14, 1431–1454.

DOR, N., RODEH, M., AND SAGIV, S. 2000. Checking cleanness in linked lists. In *Proceedings of the Static Analysis Symposium (SAS'00)*. 115–134.

DUFOUR, B., RYDER, B. G., AND SEVITSKY, G. 2007. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*. 118–128.

DUFOUR, B., RYDER, B. G., AND SEVITSKY, G. 2008. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'08)*. 59–70.

EJ-TECHNOLOGIES GMBH. 2013. JProfiler. http://www.ej-technologies.com.

GUPTA, S. C. AND PALANKI, R. 2005. Java memory leaks – Catch me if you can. http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/.

HACKETT, B. AND RUGINA, R. 2005. Region-based shape analysis with tracked locations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. 310–323.

HASTINGS, R. AND JOYCE, B. 1992. Purify: A tool for detecting memory leaks and access errors in c and c++ programs. In *Proceedings of the Winter USENIX Conference*. 125–138.

HAUSWIRTH, M. AND CHILIMBI, T. M. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. 156–164.

HEINE, D. L. AND LAM, M. S. 2003. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. 168–181.

HEINE, D. L. AND LAM, M. S. 2006. Static detection of leaks in polymorphic containers. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*. 252–261.

JIKES RESEARCH VIRTUAL MACHINE. 2013. http://jikesrvm.org.

JUMP, M. AND MCKINLEY, K. S. 2007. Cork: Dynamic memory leak detection for garbage-collected languages. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. 31–38.

JUMP, M. AND MCKINLEY, K. S. 2010. Detecting memory leaks in managed languages with cork. *Softw. Pract. Exper. 40,* 1, 1–22.

MAXWELL, E. K., BACK, G., AND RAMAKRISHNAN, N. 2010. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'10)*. 115–124.

MITCHELL, N. 2006. The runtime structure of object ownership. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06)*. 74–98.

MITCHELL, N., SCHONBERG, E., AND SEVITSKY, G. 2009. Making sense of large heaps. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'09)*. 77–97.

MITCHELL, N., SCHONBERG, E., AND SEVITSKY, G. 2010. Four trends leading to java runtime bloat. *IEEE Softw. 27,* 1, 56–63.

MITCHELL, N. AND SEVITSKY, G. 2003. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'03)*. 351–377.

MITCHELL, N. AND SEVITSKY, G. 2007. The causes of bloat, the limits of health. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07).* 245–260.

MITCHELL, N., SEVITSKY, G., AND SRINIVASAN, H. 2006. Modeling runtime behavior in framework-based applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06)*. 429–451.

NICHOLAS, E. 2006. http://weblogs.java.net/blog/enicholas/archive/2006/04/leaking evil.html.

NOVARK, G., BERGER, E. D., AND ZORN, B. G. 2009. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 397–407.

ORLOVICH, M. AND RUGINA, R. 2006. Memory leak analysis by contradiction. In *Proceedings of the Static Analysis Symposium (SAS'06)*. 405–424.

QIN, F., LU, S., AND ZHOU, Y. 2005. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'05)*. 291–302.

QUEST SOFTWARE. 2013. JProbe memory debugging. http://www.quest.com/jprobe.

RAYSIDE, D. AND MENDEL, L. 2007. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the International Conference on Automated Software Engineering (ASE'07)*. 194–203.

SHACHAM, O., VECHEV, M., AND YAHAV, E. 2009. Chameleon: Adaptive selection of collections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 408–418.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2000. Automatic removal of array memory leaks in java. In *Proceedings of the International Conference on Compiler Construction (CC'00)*. 50–66.

SHANKAR, A., ARNOLD, M., AND BODIK, R. 2008. JOLT: Lightweight dynamic analysis and removal of object churn. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*. 127–142.

STANDARD PERFORMANCE EVALUATION CORPORATION. 2000. SPECjbb2000. http://www.spec.org/jbb2000.

SUN BUG DATABASE. 2013. http://bugs.sun.com/bugdatabase.

VALLEE-RAI, R., GAGNON, E., HENDREN, L., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. 2000. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction (CC'00)*. 18–34.

XIAO, X., ZHOU, J., AND ZHANG, C. 2011. Tracking data structures for postmortem analysis. In *Proceedings of the International Conference on Software Engineering (ICSE'11)*. 896–899.

XIE, Y. AND AIKEN, A. 2005. Context- and path-sensitive memory leak detection. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'05)*. 115–125.

XU, G. 2011. Analyzing large-scale object-oriented software to find and remove runtime bloat. Ph.D. thesis, The Ohio State University.

XU, G., ARNOLD, M., MITCHELL, N., ROUNTEV, A., AND SEVITSKY, G. 2009. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 419–430.

XU, G., ARNOLD, M., MITCHELL, N., ROUNTEV, A., SCHONBERG, E., AND SEVITSKY, G. 2010a. Finding low-utility data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 174–186.

XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., AND SEVITSKY, G. 2010b. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER'10)*. 421–426.

XU, G., BOND, M. D., QIN, F., AND ROUNTEV, A. 2011. LeakChaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 270–282.

XU, G. AND ROUNTEV, A. 2008. Precise memory leak detection for java software using container profiling. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*. 151–160.

XU, G. AND ROUNTEV, A. 2010. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 160–173.

YAN, D., XU, G., AND ROUNTEV, A. 2012. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*.

Received November 2011; revised February 2012; accepted March 2012