GC Assertions: Using the Garbage Collector to Check Heap Properties

Edward Aftandilian
Tufts University
161 College Ave
Medford MA
eaftan@cs.tufts.edu

Samuel Z. Guyer
Tufts University
161 College Ave
Medford MA
sguyer@cs.tufts.edu

ABSTRACT

This paper introduces *GC assertions*, a system interface that programmers can use to check for errors, such as data structure invariant violations, and to diagnose performance problems, such as memory leaks. GC assertions are checked by the garbage collector, which is in a unique position to gather information and answer questions about the lifetime and connectivity of objects in the heap. We introduce several kinds of GC assertions, and we describe how they are implemented in the collector. We also describe our reporting mechanism, which provides a complete path through the heap to the offending objects. We show results for one type of assertion that allows the programmer to indicate that an object should be reclaimed at the next GC. We find that using this assertion we can quickly identify a memory leak and its cause with negligible overhead.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability, Statistical Methods*

General Terms

Reliability, Performance, Experimentation

Keywords

Memory Leaks, Managed Languages, Garbage collection

1. INTRODUCTION

Garbage collection provides a number of software engineering benefits, allowing programmers to avoid the pitfalls associated with manual memory management. It achieves these benefits, in part, by taking control of many aspects of memory management away from the the programmer. As result, however, programmers cannot easily check properties of objects in memory. For example, even the seemingly simple question "Will this object be reclaimed during garbage collection?" cannot easily be answered.

MSPC'08 2 March 2008, Seattle, WA, USA

The goal of this work is to develop a general mechanism, which we call a *GC assertion*, that allows the programmer to express *expected* properties of data structures and convey this information to the garbage collector. The key is that many useful, non-trivial properties can be easily checked by the garbage collector at GC time, imposing little or no run-time overhead. The collector triggers the assertion if it finds that the expected properties have been violated.

While GC assertions require extra programmer effort, they capture application-specific properties that programmers want to check and can easily express. For example, we provide a GC assertion to verify that a data structure is reclaimed at the next collection when the programmer expects it to be garbage. The programmer passes the objects to an "expect dead" assertion, and the garbage collector triggers the assertion if it encounters any of the objects during GC. The programmer cannot conveniently obtain such information any other way.

In this paper we propose a set of GC assertions designed to help identify problems in Java data structures, including memory leaks and improper structure sharing. We classify these assertions according to the property they check: lifetime and lifespan, connectivity and shape, and allocation volume. We describe ways to express these properties and combine them into more complex queries, and we describe the machinery necessary to implement different kinds of assertions. We also explore ways that the virtual machine can react to triggered assertions: the VM can report the violation and halt, or continue running; in some cases, it can attempt to enforce the property, possibly by modifying the heap. Finally, we present results for an implementation of the "expect dead" assertion, which helps to identify potential memory leaks.

The paper is organized as follows: in Section 2 we describe the kinds of assertions we support. In Section 3 we provide results for `assert-dead` using two different tracking and reporting mechanisms. We describe related work in Section 4 and outline future work and conclusions in Section 5.

2. GC ASSERTIONS

GC assertions allow programmers to validate the memory behavior of their programs by describing expected properties to the garbage collector, which automatically checks them. In this section we describe several kinds of GC assertions and how they are used by the programmer. We also describe the machinery that is necessary to check these assertions and how the collector reports the results to best aid the programmer in debugging.

The goal of GC assertions is to provide a simple and low-cost way for programmers to express and check properties of their data structures. Assertions are a familiar construct for program checks, and the kinds of assertions we support involve properties that pro-

grammers want to know about but in many cases have no direct way to check. The "expect dead" assertion, described below, checks that a given object is reclaimed at the next garbage collection. In the absence of explicit free operations, programmers have no easy way of checking that reclamation occurs, particularly at the granularity of individual object instances.

Programmers can use GC assertions to detect errors, such as memory leaks, in a way that is very different from existing techniques. Unlike in C and C++, a memory leak in Java is difficult to define: "leaked" objects are still reachable, and therefore might still be used by the program at some point in the future. Existing leak detectors, such as Sleigh and Cork [3, 9], use heuristics (such as staleness and heap growth) to identify objects that are unlikely to be used again. A GC assertion, on the other hand, captures explicit information from the programmer about future expected behavior, eliminating the need for heuristics. A violation of the assertion is an immediate and unambiguous error. For example, if the programmer asserts that an object is expected to be dead, even *accessing* that object should trigger the assertion. The downside is that adding assertions requires extra work on the part of the programmer. We believe, however, that programmers often have this knowledge and would appreciate a way to express it.

2.1 Lifetime assertions

Lifetime assertions allow the programmer to check that the lifetime characteristics of objects conform to expectations. These assertions generally work by marking the given objects at the call to the assertion and then reporting if they are encountered during collection. These lifetime properties are trivial to express and extremely cheap for the collector to check but are almost impossible to obtain by any other means. We have identified the following kinds of lifetime assertions:

`assert-dead(p)`. This assertion is triggered at the next GC if the object pointed to by *p* is not reclaimed. We implement this assertion by setting a bit in the object header and then checking for that bit during collection.

`assert-reachdead(p)`. This assertion is triggered at the next GC if any object reachable from *p* is not reclaimed. This assertion is not yet implemented, but we plan to implement it by traversing the given data structure using the existing tracing infrastructure, setting the assert-dead bit in each object's header.

`assert-alldead()`. This assertion is used in conjunction with a separate `start-region()` call: the assertion is triggered if any object allocated after `start-region()` is not reclaimed at the `assert-alldead()`. This allows the programmer to bracket a particular region of code, for example a particular method or loop body, and ensure that it is memory stable. In its simplest form, this assertion would mark objects allocated after the `start-region()` call but wait to check them until the GC immediately following the use of `assert-alldead()`. Another possibility would be to trigger a GC at the assertion, allowing more precise checking. Finally, a more complex implementation would allow multiple active regions by marking objects according to which `start-region()` call they occur in.

`assert-alldead-except(p)`. This assertion behaves like `assert-alldead()` but allows any object reachable from *p* to escape. This option is valuable for checking methods that should produce a single result, while ensuring that no intermediate data structures survive.

2.2 Volume assertions

Volume assertions express constraints on the number or total volume of particular categories of objects. These assertions are implemented by accumulating and checking volume information about categories of objects. We are currently in the process of implementing these assertions.

`assert-space(T, B)`. This assertion is triggered when the total volume of objects of type *T* exceeds *B* bytes at the next collection.

`assert-instances(T, I)`. This assertion is triggered when the total number of objects of type *T* exceeds *I* at the next collection. By passing 0 for *I*, programmers can check that no instances of a particular class exist.

`assert-maxlive(B)`. This assertion, like `assert-alldead()` above, works in conjunction with `start-region`. The assertion is triggered if the total volume of surviving objects allocated since `start-region` exceeds *B* bytes.

2.3 Shape assertions

Shape assertions allow programmers to check the connectivity properties of individual objects or data structures. Checking these assertions requires the garbage collector to compute extra information about objects, such as number of incoming pointers. We have not implemented these assertions, and some of them do not have obvious efficient implementation strategies.

`assert-unshared(p)`. This assertion is triggered if the given object has more that one incoming pointer. It is trivial to implement using the existing GC infrastructure: we set a bit in the object header, and if the GC scan code encounters the object more than once (i.e., a new pointer is found when the object is colored gray or black), we raise the assertion.

`assert-ownedby(p,q)`. This assertion is triggered if there exists a path through the heap to the object pointed to by *p* that *does not* pass through the object pointed to by *q*. This property requires both extra metadata (to store the ownership relationships) and extra work to record which owners have been encountered on a path.

2.4 Actions

When an assertion is triggered the garbage collector has several ways it can take action. The choice is made by the programmer and depends on the severity of the assertion and on whether the programmer wants the program to continue running.

Log an error, but continue executing. In the case of the lifetime assertions we can report either the reference that is directly keeping the object alive or the full path through the heap. In our experiments in Section 3 we found that our system can maintain full path information with no measurable overhead.

Log an error and halt. Similar to the case above, but is used for assertions whose failure indicates a non-recoverable error.

Force the assertion to be true. In the case of lifetime assertions, the garbage collector can force objects to be reclaimed by nulling out all incoming references. This might allow a program to run longer without running out of memory but risks introducing a null pointer exception.

We are also developing an interface to GC assertions that would allow the programmer to test the conditions directly and take action in an application-specific manner. For example, the programmer could check the live-size and exit gracefully if it appears that the program is running out of memory.

3. PRELIMINARY RESULTS

This section presents results for a simple "expect dead" assertion to test the usefulness and performance of our technique. We describe two error reporting mechanisms: one that identifies the reference directly keeping the object alive and one that provides the full path through the heap to the object. We implement this as-

sertion in Jikes RVM and show that it can quickly identify memory leaks with negligible overhead.

3.1 Implementation of expect-dead assertion

We implemented our expect-dead assertion in Jikes RVM 2.9.1 using the SemiSpace collector. We chose SemiSpace because it is a full-heap collector, which will report all assertions at every garbage collection. Our technique will work with any tracing collector, such as generational mark/sweep. A generational collector, however, performs full-heap collections infrequently, allowing some assertions to go unchecked for long periods of time.

Our implementation incurs almost no performance overhead. First, since we use an unused bit in the object header to mark objects "dead," we do not incur any space overhead. Second, the only change we make to the garbage collection algorithm is checking this bit during tracing, so we do not perturb the garbage collector and there is little-to-no runtime overhead. Thus, this assertion could be enabled in a production environment.

3.2 Providing debugging information

Once an assertion is triggered, the programmer still needs help determining the cause of the error. We present two reporting strategies that help explain to the programmer why objects that she expected to be dead are still reachable.

3.2.1 Last pointer

One simple strategy is to report the last pointer to a reachable object. We modify the GC tracing loop to store the reference currently being traced. If the assertion is triggered, we print the "dead" object's type along with the type of the object whose reference we followed to reach this object. This strategy has very low overhead, as expected (see Section 3.3 below), but produces messages of marginal value to the programmer. We found that having only the last pointer to an object made it difficult to find certain bugs. For example, consider the following warning produced during testing with SPEC JBB2000:

```
Warning: object that was asserted dead is reachable
Type: Lspec/jbb/Order;
Reachable from object of type: [Ljava/lang/Object;
```

In this case, we know that an array of Objects points to the "dead" Order object in which we are interested. However, there are many arrays of Objects in SPEC JBB2000, and it is impossible to determine exactly which one points to this object.

3.2.2 Full path

Our second, more sophisticated, reporting strategy is to provide the full path through the object graph, from root to the "dead" object. This information is extremely valuable for fixing Java memory leaks, since all leaks are ultimately caused by outstanding references to objects that are no longer needed. The full path to the leaked object identifies the reference or container that needs to be cleared to stop the leak. Our information is similar to that provided by Cork [9], but much more precise: our path consists of object instances, not just types.

Our implementation modifies the management of the worklist that holds unprocessed references for the collector during tracing (the so-called "gray" objects.) The baseline algorithm performs a depth-first search by popping a reference off the worklist, scanning the object, and pushing all its outgoing references back on the worklist. In our algorithm, we keep this object on the worklist while its outgoing references are being traced, allowing us to reconstruct the path when necessary. We pop a reference from the worklist, set its low order bit and push it back onto the worklist; then we continue to scan the object normally. Because all objects in Jikes RVM are word aligned, the two low order bits are unused, and we can safely use one of them for this algorithm. If we encounter a reference whose low-order bit is set, we discard it and continue – this simply indicates that we have already visited all objects reachable from it. Thus, at any given time during tracing, the subset of the worklist whose references have their low bit set define the complete path from the root to the current object. The full-path output for the SPEC JBB2000 assertion above is as follows:

```
Warning: an object that was freed is reachable.
Type: Lspec/jbb/Order;
Path to object:  Lspec/jbb/Company; ->
  [Ljava/lang/Object; ->
  Lspec/jbb/Warehouse; ->
  [Ljava/lang/Object; ->
  Lspec/jbb/District; ->
  Lspec/jbb/infra/Collections/longBTree; ->
  Lspec/jbb/infra/Collections/longBTreeNode; ->
  [Ljava/lang/Object; ->
  Lspec/jbb/infra/Collections/longBTreeNode; ->
  [Ljava/lang/Object; ->
  Lspec/jbb/Order;
```

3.3 Performance

3.3.1 Methodology

We implemented GC assertions on top of Jikes RVM 2.9.1, using the SemiSpace collector. We measure performance for the DaCapo benchmarks (2006-10-MR2) [2], SPEC JVM98 [16], and a fixed-workload version of SPEC JBB2000 called `pseudojbb` [17]. We report total execution time and GC time; mutator times were unchanged, so we do not report them. For SPEC JVM98, we use the large input size (`-s100`); for DaCapo and `pseudojbb`, we use the default input size. (Note: we omit results from DaCapo `chart` because we could not get it to run on the machine we use for benchmarking.) While none of our benchmarks contain expect-dead assertions, these experiments measure the overhead of checking the extra bits and recording debugging information. When warnings are triggered, we would expect our system to have greater overhead. All experiments are run on a 2.0 GHz Pentium-M machine with 2 GB of RAM, running Linux 2.6.20.

We use the adaptive configuration of Jikes RVM, which dynamically identifies frequently executed methods and recompiles them at higher optimization levels. For the DaCapo benchmarks, we use the built-in `-converge` flag to run each benchmark until its times converge; we repeat this five times for each benchmark and average the results. For `pseudojbb` and the SPEC JVM98 benchmarks, we imitate the DaCapo converge option using scripts.

We executed each benchmark with a heap size fixed at two times the minimum possible for that benchmark using the SemiSpace collector. We provide the heap sizes we used in Table 1.

3.3.2 Discussion

Our results in Table 1 show that the expect-dead assertion does not affect overall performance on "correct" code. Surprisingly, the last-pointer reporting technique shows higher overhead than the full-path technique. We see a 7.43% increase in garbage collection time for the last-pointer technique versus either the unmodified system or the full-path configuration. However, the overall execution time increases by only 1.53%. Surprisingly, the full-path reporting technique has even lower overhead – a 0.64% *increase* in performance, which is experimental noise.

		Baseline (stddev) (ms)		Last pointer (stddev) (% over)		Full path (stddev) (% over)	
Benchmark	Heap (MB)	Total	GC	Total	GC	Total	GC
compress	48	4234 (10.8)	295 (0.2)	0.93% (34.7)	13.02% (0.2)	0.03% (15.1)	2.13% (0.5)
jess	64	2289 (33.3)	787 (0.5)	4.13% (21.8)	11.75% (1.9)	0.27% (38.5)	0.11% (0.6)
raytrace	48	1988 (35.8)	797 (2.9)	1.09% (26.5)	9.22% (0.4)	-1.47% (55.6)	1.89% (2.6)
db	64	10139 (193.2)	445 (2.7)	0.40% (212.5)	8.68% (0.2)	0.45% (197.3)	2.65% (0.3)
javac	64	4669 (47.2)	1373 (5.8)	2.99% (43.0)	10.67% (10.3)	0.79% (35.1)	2.49% (7.2)
mpegaudio	32	4151 (29.4)	198 (24.1)	-0.49% (86.0)	6.60% (19.9)	-0.87% (39.2)	-3.80% (18.5)
mtrt	64	2072 (42.6)	773 (12.2)	2.85% (85.7)	7.95% (41.1)	3.26% (59.2)	3.59% (12.0)
jack	48	3472 (136.5)	1038 (42.4)	3.17% (107.2)	14.17% (47.0)	-1.70% (22.6)	1.59% (23.5)
pseudojbb	160	5501 (14.4)	620 (5.5)	2.50% (150.5)	7.54% (64.6)	1.30% (28.0)	-1.35% (4.4)
antlr	64	3752 (68.9)	1060 (45.8)	1.49% (74.2)	5.23% (46.7)	1.22% (90.6)	0.79% (47.9)
bloat	112	13689 (448.1)	4634 (209.4)	-1.40% (190.0)	1.01% (107.2)	-4.50% (111.6)	-7.31% (55.4)
eclipse	224	49380 (1036.2)	11811 (133.0)	-0.90% (1004.5)	6.85% (212.2)	-1.84% (1212.1)	-1.55% (278.6)
fop	112	2071 (47.6)	167 (6.0)	2.76% (58.4)	7.49% (6.6)	0.36% (19.5)	-2.24% (1.0)
hsqldb	352	5648 (30.3)	3761 (5.0)	0.67% (60.9)	1.62% (1.7)	-2.67% (30.2)	-3.51% (11.1)
jython	144	13459 (633.9)	4172 (214.8)	1.03% (304.4)	5.46% (202.2)	-2.99% (588.7)	-5.97% (246.6)
luindex	64	12189 (26.2)	2193 (41.8)	1.29% (103.4)	7.26% (46.1)	0.51% (126.2)	0.73% (34.2)
lusearch	96	18592 (382.7)	6206 (64.9)	3.50% (301.4)	7.72% (63.8)	1.20% (608.9)	1.69% (243.2)
pmd	128	8879 (83.4)	3103 (25.7)	2.05% (77.5)	4.77% (37.9)	-1.36% (66.0)	-3.98% (62.0)
xalan	112	16001 (257.7)	4778 (229.0)	1.22% (193.5)	5.07% (270.7)	-3.76% (199.6)	-7.65% (229.1)
mean				1.53%	7.43%	-0.64%	-1.10%

Table 1: Run-time overhead for two different assertion reporting schemes: reporting the last pointer to an object and reporting the full path to an object. The first set of benchmarks belong to SPEC JVM98, the second `pseudojbb`, and the third the DaCapo Benchmark Suite. Times for the baseline configuration are in milliseconds; values for the last pointer and full path configurations are given in terms of percent overhead compared to the baseline configuration. Standard deviations in milliseconds are provided in parentheses after each value.

3.4 Qualititative evaluation

In addition to the benchmarks above, we tested our expect-dead assertion on SPEC JBB2000 to search for memory leaks and other errors. SPEC JBB2000 is a benchmark that measures the performance of server-side Java. It emulates a three-tier business system, with data stored in binary trees rather than an external database.

SPEC JBB2000 uses the Factory pattern to create and dispose of objects. We instrumented the `destroy()` method of the Entity object with an expect-dead assertion, believing that an object that had been destroyed should be unreachable.

We first found that "dead" Order objects were reachable from Customer objects. Upon further investigation, we found that each Customer object maintains a reference to the last Order this Customer placed. When the Order was destroyed, the `lastOrder` field in the associated Customer was not cleared, and this reference prevented the Order from being reclaimed. Since each Order object maintains a reference to the Customer to which it belongs, we were able to repair this leak by setting the reference in the Customer to null when the Order is destroyed. We found a similar situation with Address objects, which were also pointed to by Customer objects, but we were not able to repair it since there is no back reference from Addresses to Customers.

The second problem we found was more subtle. In the main loop of the benchmark, the Company object from the previous iteration is destroyed before creating the Company object for the current iteration. The previous company is referenced in the `oldCompany` local variable, which remains visible through the whole method, which means that the previous Company object cannot be reclaimed. Simply setting the variable to null after the Company is destroyed allows this whole Company data structure to be reclaimed earlier, potentially allowing the program to run in less memory.

Finally, we investigated a known memory leak in SPEC JBB2000 first reported by Jump and McKinley [9]. SPEC JBB2000 places Order objects into an `orderTable`, implemented as a BTree, when they are created. They are completed during a DeliveryTransaction but are not removed from the table, causing a memory leak. To find this leak, we placed an expect-dead assertion for the Order object at the end of `DeliveryTransaction.process()`. Our GC assertions system showed us the path through the object graph where these Order objects were reachable, and with this information we were able to repair the leak. It is important to note that, for the GC assertion to work, the programmer must know that the Order object should be dead at the end of `DeliveryTransaction.process()`. However, in a large project where no single programmer can understand the whole system, a GC assertion like this would be helpful in explaining anomalous behavior.

4. RELATED WORK

Our work is most closely related to other run-time techniques for detecting memory and data structure errors. Most of this work differs from ours in that it relies primarily on general rules of thumb and statistical anomalies to detect errors. Leak detectors, for example, are often based on the assumption that objects that have not been accessed for a long time are a likely memory leak. GC assertions, on the other hand, check specific properties supplied by the programmer. While they detect only those problems for which the programmer has added assertions, an assertion violation is an unambiguous event: the program is definitely behaving in a way that the programmer considers incorrect.

4.1 Leak detection

A number of systems have been designed to detect memory leaks, both in managed and unmanaged languages. The challenge in managed languages is determining what constitutes a leak, since leaked objects are still reachable. Some tools use the notion of "staleness": objects that have not been accessed in a "long time" are probably memory leaks [6, 3]. Other tools use heap differencing to find objects responsible for heap growth [1, 14, 13, 10, 9]. GC assertions, however, provide a way for programmers to provide explicit knowledge about which objects should be dead. This information, when available, is not a heuristic: assertion violations are never false positives, and they indicate a precise and specific error condition.

4.2 Data structure checking

Previous work on data structure checking has utilized both dynamic and static techniques. HeapMD [5] monitors properties of objects (such as in-degree and out-degree) at run time and reports anomalies as possible errors. Static approaches include various

techniques for static analysis of heap structures, including pointer analysis and shape analysis [4, 8, 15]. As with leak detection, the primary difference between this work and ours is that we allow the programmer to declare explicitly what conditions constitute an error, and we check those conditions precisely and cheaply.

ESC/Java [7] allows programmers to declare some properties of their data structures, such as member fields that cannot be null. This capability is similar in spirit to GC assertions, but is enforced by static checking, and therefore is limited by conservative analysis in the same ways that other static analysis techniques are limited.

4.3 User-controlled garbage collection

The Java programming language provides a standard library call that invokes the garbage collector: `System.gc()`. What this call does, however, is left entirely up to the JVM implementor. For example, in Sun's Hotspot JVM, `System.gc()` performs a full-heap collection, but in BEA's JRockit, it performs a nursery collection only. Our goal is to provide a more systematic and rich interface for programmers to interact with the garbage collector and other components of the JVM.

More recently, Sun introduced the JVM Tool Interface (JVMTI) to the JVM specification to allow tool developers to monitor GC activity. Many of our GC assertions could be implemented using JVMTI, with the advantage that it is (mostly) portable across different JVMs. We chose not to use JVMTI for two reasons. First, many of the hooks we need for GC assertions are optional parts of the spec. Second, we wanted to experiment with reporting mechanisms, such as the full object path, that are not supported by JVMTI. Modifying the virtual machine is also likely to incur a lower performance overhead.

O'Neill and Burton propose a mechanism that allows users to annotate objects with small pieces of code called *simplifiers*, which are executed by the garbage collector [12]. Simplifiers provide a general mechanism for injecting arbitrary user code into the GC process, and are primarily focused on performance. It might be possible to implement ad-hoc GC assertions on top of simplifiers by adding an explicit flag to each class along with a simplifier that checks the flag. However, it would be difficult to provide the debugging information, such as the full object path.

The COLA system allows programmers to dictate the layout order of objects to the garbage collector using an iterator-style interface [11]. Like simplifiers, the focus of COLA is on controlling the garbage collector's behavior to improve performance.

5. CONCLUSIONS AND FUTURE WORK

The garbage collector is a powerful source of information about program state and behavior: it systematically visits all objects and all references in the heap (in the case of a full-heap collection). It is in a unique position to check and report a variety of programmer-supplied assertions about data structures. Furthermore, the garbage collector can check properties, such as object lifetime, that no other subsystem has access to. This paper represents a first step towards taking advantage of these capabilities by giving programmers a structured way to communicate with the garbage collector.

Our future work will explore the following: (1) Implement the remaining GC assertions described in this paper, assess their usefulness and measure their overhead. (2) Explore more sophisticated techniques for specifying and testing assertion conditions, such as an API that programmers can use to monitor or query the garbage collector. (3) Experiment with assertion enforcement: in a production system, it may be desirable to enforce the assertion rather than to issue an error message. (4) Investigate assertions based on other run-time systems: are there useful questions that other run-time system components could easily answer? What information might the programmer want to know from the thread scheduler or the just-in-time compiler?

6. REFERENCES

[1] BEA. JRockit Mission Control. http://dev2dev.bea.com/jrockit/tools.html.

[2] S. M. e. a. Blackburn. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[3] M. D. Bond and K. S. McKinley. Bell: Bit-Encoding Online Memory Leak Detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, New York, NY, USA, 1990. ACM.

[5] T. M. Chilimbi and V. Ganapathy. HeapMD: Identifying Heap-based Bugs using Anomaly Detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[6] T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming language design and implementation*, pages 234–245, 2002.

[8] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[9] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *Symposium on Principles of Programming Languages*, pages 31–38, 2007.

[10] N. Mitchell and G. Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming*, pages 351–377, 2003.

[11] G. Novark, T. Strohman, and E. D. Berger. Custom object layout for garbage-collected languages. Technical Report UM-CS-2006-06, UMass Amherst, 2006.

[12] M. E. O'Neill and F. W. Burton. Smarter garbage collection with simplifiers. In *Workshop on Memory System Performance and Correctness*, pages 19–30, 2006.

[13] Quest. JProbe Memory Debugger. http://www.quest.com/jprobe/debugger.asp.

[14] SciTech Software. .NET Memory Profiler. http://www.scitech.se/memprofiler/.

[15] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with application to compile-time memory management. In *Static Analysis Symposium*, pages 483–503, San Diego, CA, June 2003.

[16] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.

[17] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.