GC Assertions: Using the Garbage Collector to Check Heap Properties

Edward E. Aftandilian Samuel Z. Guyer

Department of Computer Science
Tufts University
{eaftan,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. By piggybacking on existing garbage collector computations, our system is able to check heap properties with very low overhead – around 3% of total execution time – low enough for use in a deployed setting.

We introduce several kinds of GC assertions and 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 report results on both the performance of our system and the experience of using our assertions to find and repair errors in real-world programs.

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 many software engineering benefits, most notably by eliminating a large class of insidious programming errors associated with manual memory management, such as dangling pointers and double frees. One downside of automatic memory management, however, is that programmers are left with less control and less information about the memory behavior of their programs. For example, in the absence of explicit free operations, Java programmers have no way to answer even seemingly simple questions, such as "Will this object be reclaimed during the next garbage collection?"

In this paper we present *GC assertions*, an introspective interface that allows programmers to query the garbage collector about the run-time heap structure and behavior of their programs. Like ordinary assertions, programmers add GC assertions to their code to express expected properties of objects and data structures. Unlike ordinary assertions, however, GC assertions are not checked immediately. Instead, when GC assertions are executed they convey their information to the garbage collector, which checks them during the next collection cycle. The key to our technique is that we piggyback these checks on the normal GC tracing process, imposing little or no additional cost.

We describe a suite of GC assertions designed to help identify bugs in Java data structures, including memory leaks and improper structure sharing. Our selection of assertions balances two competing goals. The first is to provide a rich and expressive set of assertions that programmers find easy to use and valuable. The second is to keep the run-time overhead low enough that the system can be used to detect and prevent errors in deployed software. To this end, we have identified several categories of heap properties, including lifetime and lifespan, allocation volume, and connectivity and ownership, that can be checked during a single pass over the heap. In addition, we minimize space overhead by limiting our meta-data to the set of assertions to be checked and extra bits stolen from object headers. Even with a significant set of assertions to check during each garbage collection, our technique increases collection time by less than 14% and overall runtime by less than 3%.

Using the garbage collector to check programmer-written heap assertions provides a combination of features not available with existing techniques:

PLDI'09, June 15–20, 2009, Dublin, Ireland.

- *More precise than static analysis.* Unlike static heap checking, which operates on an abstract model of the heap, our technique works on the actual concrete heap, avoiding the need for conservative approximations. In addition, our technique is unaffected by features that typically thwart static analysis, such as dynamic class loading, reflection, and bytecode rewriting.
- *More efficient than run-time invariant checking.* Systems that support true program invariants must check these properties after any computation that might violate them, imposing a substantial performance penalty – as much as 10X to 100X slowdown [12]. By deferring checks until garbage collection time, our system can batch them together and leverage the existing collection computation to eliminate much of the overhead. The price we pay is that we can miss a transient error if it does not persist across a GC cycle.
- *More accurate than heuristics.* Unlike tools based on heuristics (such as "staleness") or anomaly detection, GC assertions capture information that is both precise and application specific. Although adding assertions requires extra effort on the part of programmers, the system generates no false positives because any violation represents a mismatch between the programmer's expectations and the actual program behavior.

We describe our implementation of these assertions in the JikesRVM virtual machine. We also explore different ways that the virtual machine can react to triggered assertions and the kinds of information it can provide to help the programmer diagnose the bug. Finally, we present results for real programs, showing both the debugging experience and the performance impact.

The rest of this paper is organized as follows: in Section 2 we describe the kinds of assertions we support and their implementation in the Jikes RVM research virtual machine. In Section 3 we provide both quantitative performance results and qualitative debugging results. We describe related work in Section 4 and conclusions in Section 5.

2. GC Assertions

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 program behavior that programmers want to know about, but in many cases have no direct way to check. The "assert 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.

In this section we describe the specific GC assertions we support, how they can be used by the programmer to detect errors, and how the assertions are implemented in the Jikes RVM research virtual machine. We also discuss different ways that the system can react to assertion violations, since they are not detected at the point the assertion is executed.

2.1 Adding assertions

Unlike other approaches, our technique is programmer-driven: it requires extra effort on the part of the programmer to add assertions. This design, however, has several advantages over tools based on heuristics or anomaly detection. First, GC assertions capture properties that programmers already know and want to be able to express. Second, these properties often represent high-level information that cannot be inferred from the program. Because this information is precise and application-specific, any violation of a GC assertion is an immediate and unambiguous error.

For example, we can use GC assertions to detect memory leaks in a way that is very different from previous techniques. 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. Without extra information, most leak detectors must rely on heuristics to identify potential leaks. Some tools use the notion of "staleness" to identify potential leaks: objects that have not been accessed in a "long time" are probably memory leaks [14, 7]. Other tools use heap differencing to find objects that are probably responsible for heap growth [3, 37, 35, 32, 27]. These techniques, however, can only suggest *potential* leaks, which the programmer must then examine manually. Using GC assertions programmers can tell our system exactly when particular objects should be dead. Violations can be detected almost immediately, rather than having to wait for objects to become stale or fill up the heap. Our system provides detailed information about the reason for the failure (for example, the path through the heap that is keeping the object alive.)

2.2 Implementation overview

We implemented these assertions in Jikes RVM 3.0.0 using the MarkSweep collector. We chose MarkSweep because it is a full-heap collector, which will check 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.

2.3 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. Lifetime properties are easy to express and extremely cheap for the collector to check,

but are almost impossible to obtain by any other means. We have developed the following kinds of lifetime assertions:

2.3.1 assert-dead

`assert-dead(p)` is triggered at the next garbage collection if the object pointed to by p is not reclaimed (i.e., found to be still reachable.)

Usage. `assert-dead` allows the programmer to check that a particular object is reclaimed at or soon after a specific point in the program. For example, a common Java idiom is to assign null to a reference when the object pointed to should be reclaimed. However, if there are still other references to the object, the object will not be reclaimed. `assert-dead` can be used in this situation to verify that the object is reclaimed.

Implementation. We implement assert-dead as follows. During execution, when the JVM encounters this assertion, it marks the object pointed to by p as "dead" using a spare bit in the object's header. At the next garbage collection, the garbage collector checks whether any object encountered during tracing has its "dead" bit set. If so, it prints a warning along with debugging information to help the programmer find the error that led to this "dead" object being reachable.

Because we use spare bits in object headers to store information about which objects are expected to be dead, there is no space overhead for this assertion. Time overhead is limited to checking the state of a bit in the object's header when it is encountered during GC. Because the object's header must be read (and possibly written) anyway as part of the GC tracing process, the data is already in cache and the slowdown is minimal.

2.3.2 assert-alldead

`assert-alldead()` 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. The region is confined to a single thread (i.e. each thread can independently be either in or out of a region).

Usage. This assertion is useful to ensure that certain regions of code do not "leak" memory into other parts of the application. For example, in a server application, one might bracket the connection servicing code with `start-region` and `assert-alldead` assertions to ensure that, when the server has finished servicing the connection, all memory related to that connection is released.

Regions are widely used in the C/C++ world [4, 41], particularly by the Apache HTTP Server [18] and other projects that use the Apache Portable Runtime [20]. Rather than enforce region behavior to improve performance, our assertions *check* for region behavior in order to validate programmer expectations.

Implementation. We implement `assert-alldead` as follows. Each thread in Jikes RVM has a boolean flag to indicate whether it is currently in an `alldead` region, as well as a queue to store a list of objects that have been allocated while in the current region. When the programmer invokes the `start-region` assertion, the flag in the thread is switched to indicate that we are now in a region. Every allocation checks the flag to determine if it occurred within a region, and if it is, the allocated object is added to the queue. When the `assert-alldead` call occurs, the region flag is reset and the queue is processed, calling `assert-dead` on each object in the queue.

The space overhead for this assertion depends on whether a region is currently active in the given thread. If not, the space overhead is a boolean and a queue reference for each thread. Otherwise, the space overhead consists of a boolean and a queue for each region that is currently active, plus a word for each object that has been allocate while the region has been active. When the region ends, the queue will be flushed, and we will reclaim the one-word-per-object space needed for the region metadata.

The time overhead consists of checking the thread's region flag on every allocation, plus, if we are in an active region, adding the newly allocated object to the thread's region object list. By using `assert-dead` to mark the objects at the end of the region, we do not incur an extra time or space penalty to check that objects are deallocated at the end of the region.

2.4 Volume assertions

Volume assertions express constraints on the number or total volume of particular categories of objects. These assertions are implemented by accumulating the information during heap scanning, and checking against the constraints when finished.

2.4.1 assert-instances

`assert-instances(T, I)` 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.

Usage. This assertion is most useful for checking that either 0 or 1 instances of a certain class exist. For example, one might use this assertion to check that the singleton pattern is being correctly followed. For a variety of reasons, including subclassing and serialization, this design pattern is difficult to implement correctly [22]. With GC assertions, we can easily check for correctness by asserting that only one instance of the class exists at a time. Note, however, that we cannot *enforce* the singleton pattern using GC assertions.

Another potential use is in situations where the number of objects of a certain type should be limited for best performance, but it is not strictly *an error* if the limit is exceeded. We discuss such a situation in Section 3.2.2.

Implementation. Our implementation of assert-instances is different from that of the previous two assertions since this assertion is tied not to object instances but to types. In Jikes RVM, the RVMClass class corresponds to a Java class, so we modify RVMClass to maintain two extra pieces of information: the instance limit and the instance count for this class.

When the virtual machine encounters an `assert-instances` call, we set the instance limit for the type and add the type to a list of types for which we are tracking instances. During GC, every time we encounter an object of a tracked type, we increment the corresponding RVMClass's instance count. At the end of GC, we iterate through our list of tracked types, checking whether the instance limit has been violated. If so, we print a warning for the user.

This implementation incurs a space overhead of two words per loaded class (for the instance limit and instance count), as well as one word per tracked type (i.e. a type that has had an instance limit asserted) for the array of tracked types. We also incur a small time overhead by checking the RVMClass of every object during tracing, plus incrementing the instance count if necessary and checking the list of tracked types for violations at the end of GC.

2.5 Ownership assertions

Ownership assertions allow programmers to check the connectivity properties of individual objects or data structures. The garbage collector is in a unique position to check such properties, since it traverses all reachable objects in the heap, regardless of their type or access control qualifiers.

2.5.1 assert-unshared

`assert-unshared(p)` is triggered if the given object has more than one incoming pointer. It is a simple test to ensure than an object has no more than one direct parent.

Usage. This assertion can be used to check simple connectivity properties of data structures. For example, one can use `assert-unshared` to verify that a tree data structure has not inadvertently become a DAG.

Implementation. Our implementation of `assert-unshared` is similar to that of `assert-dead`. Once again, the programmer must assert in the program code that a object should be unshared after a certain point. The JVM marks the object as "unshared" by setting a spare bit in the object header. During garbage collection, the garbage collector checks objects that are encountered more than once (i.e. whose mark bits are already set) for this "unshared" bit. If such an object is encountered, we print a warning along with debugging information.

There is no space overhead for this assertion since we use a spare bit in the object header, and the time overhead is just the cost of checking the bit in each object's header during GC tracing.

2.5.2 assert-ownedby

`assert-ownedby(p,q)` is triggered if the object pointed to by q is not owned by the object pointed to by p.

There are several different ways to define what it means for one object (the *owner*) to own another object (the *ownee*) [15, 10]. Initially, our ownership assertion required that all paths through the heap from the roots (local and global variables) to the ownee must pass through the owner. This definition, however, is too restrictive to be practical: common constructs, such as iterators, violate the assertions and make them useless. Instead we provide a notion of ownership that focuses on detecting unexpected structure sharing, particularly when it impacts object lifetimes. Our definition is as follows: once ownership is asserted, the set of paths through the heap to the ownee must include *at least one path* that passes through the owner. The idea is that an ownee may be referenced by other objects, but it should never outlive its owner. This property is checked for each owner/ownee pair at every garbage collection.

As we show in Section 3, this assertion is often a more natural way to find memory leaks than using `assert-dead(p)`. Instead of identifying the point at which an object is no longer needed, the programmer just identifies the larger data structure that governs its lifetime. For example, consider a data structure in which elements are stored in a main container and also cached in a hash table. We can assert that the container owns the elements; if the system ever finds elements that are only reachable from the hash table, it reports an error.

We impose some restrictions on ownership in order to keep the cost low: the regions of the heap governed by different owner objects may not overlap. That is, the path from an owner to its ownee should not pass though any other owner (or its ownees). We discuss this restriction further below.

Usage. We expect `assert-ownedby` to be most useful when an object's lifetime is correlated to the lifetime of its owner collection. That is, when an object should not outlive its owner collection or survive when removed from that collection. For example, an order processing program might store orders in a collection, and when those orders have been processed, they are removed from the table and should be deallocated. Using `assert-ownedby` to assert that the orders are owned by their collection would help the programmer detect memory leaks caused by outstanding references to these order objects.

Implementation. `assert-ownedBy` is the most complicated assertion to implement. Our goal is to check this assertion with no extra GC work and without storing extraneous path information during collection.

With `assert-ownedBy`, the user expresses owner/ownee pairs. There may be an unbounded number of owners and ownees, and we wish to check them all in a single GC pass. In its most general form, this problem incurs a significant overhead in space and time. Consider the following general algorithm for checking ownership assertions: during GC tracing, if the collector encounters an ownee, it checks to see if that ownee's owner is on this path. If so, the ownee is marked as "owned." The other possibility is that the collector encounters the owner and then a previously marked object. We need to know whether the ownee is reachable from this previously marked object, but we do not want to repeat the tracing work. One way around this would be to bubble ownee information up the path when an ownee is encountered. In the general case, this results in each object being tagged with all ownees reachable from it. The space and time overhead from storing this information is prohibitive.

To avoid this problem, we modified the garbage collector to trace objects in a different order. Instead of starting at the roots, we added a new "ownership" phase to the collector that starts tracing from each owner object. If we encounter an ownee object that belongs to the current owner, we mark it as "owned." After tracing from all the owners, we enter the standard root scanning phase and allow the collector to proceed as normal. If the GC encounters an ownee object that has not been marked as "owned," we know it is not reachable from its owner, and we print a warning and debugging information. Notice that the portions of the heap that are reachable from the owners are marked in the ownership phase, so they will not be traced again. Thus we are able to check the ownership assertion without per-object memory overhead or processing any objects twice.

This strategy solves the performance problem, but . First, by starting GC tracing with the owner objects, we are assuming they are live. This may not be the case. To address this problem, we avoid marking the owner object when we do the ownership scan. We still mark all objects reachable from the owner. For the owner to remain live, it needs to be marked during the root scan phase, that is, it must be reachable from a root. Thus if the owner object is unreachable, it will be collected during this GC. However, any objects reachable from the owner that are not reachable from a root will not be collected until the next GC. This results in additional memory pressure that may cause the next GC to occur sooner.

The second problem relates to data structure overlap. Suppose an ownee object is reachable from both its owner and another object's owner, and the other owner is selected to be traced first in the ownership phase. The ownee will be marked but not set as "owned" because we did not start scanning from its owner. When we start tracing from the ownee's owner, the ownee has already been marked and will not be processed again. Thus we will trigger a false warning for this object. One could address this problem by enforcing a condition that the data structures defined by owners be *disjoint*. However, data structures in real problems are usually not disjoint; their objects often have back edges that result in significant overlap. Instead, we designed the ownership scanning phase to stop following a path when an ownee is reached. Ownees are added to a queue and processed after the scanning from owners has been completed. Thus collections are essentially truncated when their leaves are reached, avoiding the back edge problem. We do enforce a condition that the parts of the data structure from the owner node to the ownees must be disjoint from that of other owners. For the typical use-case where `assert-ownedBy` is used to keep track of objects in a collection, this design maintains the desired semantics, while supporting the kinds of data structures that occur in real programs.

To check this assertion we must maintain a list of owner-ownee pairs. We implement this as a pair of arrays, one containing owner objects and the other containing arrays of ownee objects, one for each owner. Thus the metadata overhead for this implementation is one word per owner or ownee object. Time overhead is as follows: for each ownee object encountered during tracing, we must check an ownee array to see if it owned by the correct owner. The ownee arrays are sorted, so we do a binary search to find the ownee object. Thus, the worse case time overhead is $n \log n$, where n is the number of ownee objects. In practice we find the overhead to be negligible, as discussed in Section 3.1.

2.6 Assertion violations

When an assertion is triggered the garbage collector has several ways it can take action.

- **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.

In this system, we choose to log the error and continue executing, so that we retain the semantics of the program without any assertions. We discuss our error reporting scheme in Section 2.7 below.

```
Warning: an object that was asserted dead 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;
```

Figure 1. Example of full-path error reporting. Each line gives the type of an object along the path from root to the object of interest.

However, for future work we would like to explore the other options above, as well as a programmatic interface that would allow the programmer to test the conditions directly and take action in an application-specific manner. It might make sense to support different actions based on the class of assertion that is violated.

2.7 Providing debugging information

Once an assertion is triggered, the programmer still needs help determining the cause of the error. For most of these assertions, the problem occurs because of an unexpected path through the heap to the offending object. Thus, displaying that path for the user would be the best way to help pinpoint the error.

Our 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 [27], 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. Figure 1 shows an example of the full-path output provided when an assertion violation is detected.

A limitation of this technique is that, to print this information for the user, we must be able to identify the offending object or path when we first encounter it. For assert-dead and assert-ownedBy, the detection algorithm naturally provides this information. However, for assert-unshared, we have no way of knowing which path is the "correct" one, and we only know there is a problem when we encounter the second path. We can print the second path, but it may not help the user find the problem. Similarly, with assert-instances, we only know that there is a problem after we have exceeded the instance limit for a type, and the "problem" paths may have been traced earlier. In these cases, the user will need to use other tools if she cannot find the problem with the given debugging output.

3. Results

We implemented GC assertions in Jikes RVM 3.0.0 using the MarkSweep collector. This section describes our results, presenting performance results for the system along a description of our experiences using it to find and fix bugs. We collected performance measurements using a standard benchmark suite; for our qualitative evaluation we used GC assertions to check for errors in real-world programs.

3.1 Performance

We first present performance measurements for GC assertions running on a standard set of benchmarks. For most of the benchmarks we measure the performance of running with no assertions, in order to determine the baseline cost of adding the assertion infrastructure into the collector. Due to the effort required to add assertions to unfamiliar code, we present measurements for two of the benchmarks running with a non-trivial set of assertions added. In all cases, the overhead of the system remains extremely low, typically around 3% or 4%.

3.1.1 Methodology

We use the DaCapo benchmarks (2006-10-MR2) [6], SPEC JVM98 [39], and a fixed-workload version of SPEC JBB2000 called `pseudojbb` [40] to quantify performance. For SPEC JVM98, we use the large input size (`-s100`); for DaCapo and `pseudojbb`, we use the default input size. All experiments were 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. We iterate each

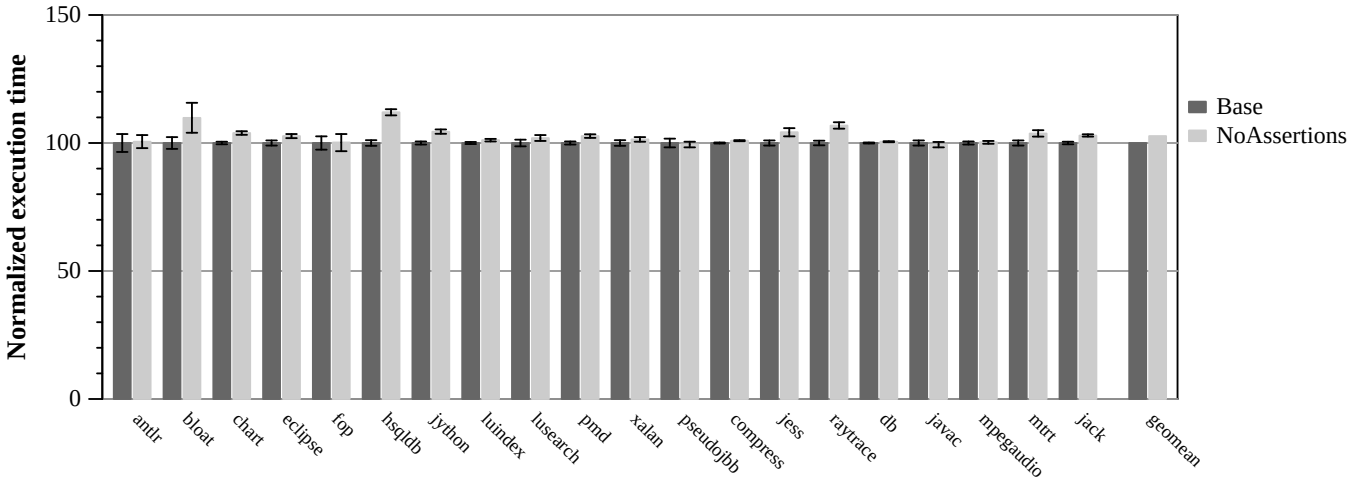

Figure 2. Run-time overhead for GC assertion infrastructure. The Base configuration corresponds to the unmodified JikesRVM. The NoAssertions configuration corresponds to a modified JikesRVM that supports GC assertions running benchmark code with no assertions.

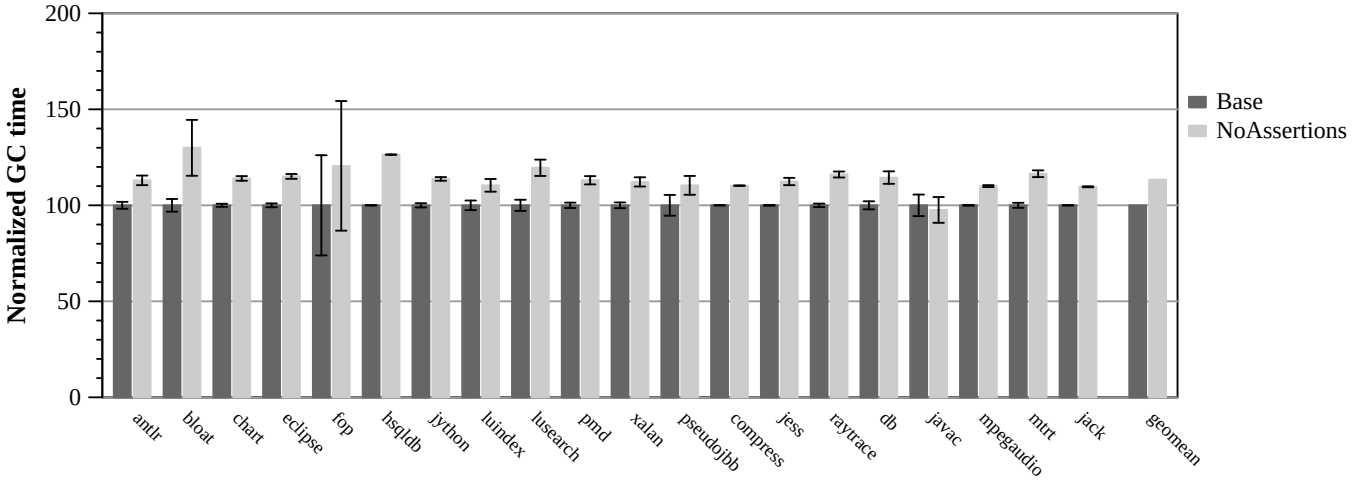

Figure 3. GC-time overhead for GC assertion infrastructure. The Base configuration corresponds to the unmodified JikesRVM. The NoAssertions configuration corresponds to a modified JikesRVM that supports GC assertions running benchmark code with no assertions.

benchmark four times and record the results from the fourth iteration. We repeat this twenty times for each benchmark.

We execute each benchmark with a heap size fixed at two times the minimum possible for that benchmark using the MarkSweep collector.

In Figures 2 and 3 we report two results for each benchmark. The Base configuration corresponds to running the unmodified benchmark on an unmodified version of Jikes RVM 3.0.0, using the MarkSweep collector. The NoAssertions configuration runs the unmodified benchmark on our modified version of Jikes RVM that supports GC assertions. This experiment measures the overhead of checking the extra bits and recording debugging information. We report the change in total execution time and GC time separately so the reader can understand the performance impact on both overall execution time and the GC subsystem. The error bars correspond to a 90% confidence interval.

In Figures 4 and 5 we report results for `_209_db` and `pseudojbb` where we modified the benchmarks to include GC assertions in appropriate places. For example, in `_209_db` we asserted that all Entry objects are owned by their containing Database object, and we added assert-dead assertions at code locations where the authors had assigned null to an instance variable (a common Java idiom that usually indicates that the object pointed to should be unreachable). We describe our modifications to `pseudojbb` in Section 3.2.1 below. These tests are meant to simulate the typical usage of GC assertions in production code. Again, the error bars correspond to a 90% confidence interval.

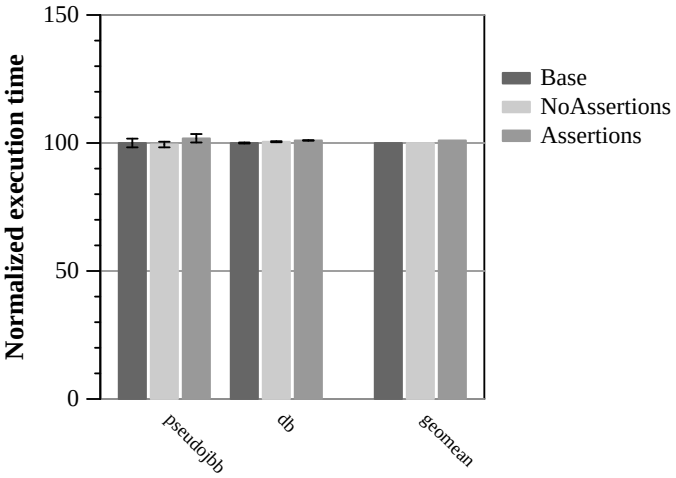

Figure 4. Run-time overhead for GC assertion checking. This set of benchmarks shows the impact on total run-time of running code that includes GC assertions and checking them at runtime. The Assertions configuration corresponds to a modified Jikes RVM that supports GC assertions running modified benchmarks that include GC assertions.

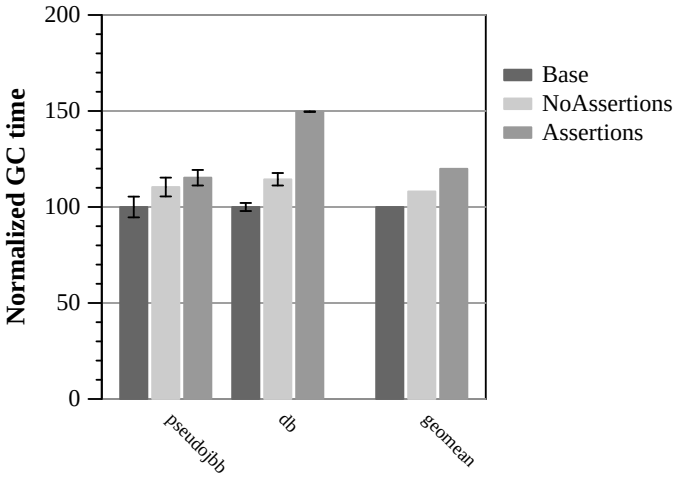

Figure 5. GC-time overhead for GC assertion checking. This set of benchmarks shows the impact on GC time of running code that includes GC assertions and checking them at runtime. The Assertions configuration corresponds to a modified Jikes RVM that supports GC assertions running modified benchmarks that include GC assertions.

3.1.2 Discussion

For the NoAssertions configuration, our results in Figures 2 3 show that the overhead of the assertion-checking infrastructure is negligible. Overall execution time increases by 2.75%, and mutator time is essentially unchanged at 1.12%. GC time increases by 13.36%, which is reasonable considering that the collector must perform several checks on every object it encounters.

For the Assertions test, our results in Figures 4 and 5 show that using GC assertions in the benchmark code has a negligible effect on performance compared to the NoAssertions case. For `_209_db`, running time increases by 0.47% and GC time by 30.1%, and for `pseudojbb`, running time increases by 2.47% and GC time by 4.40%. Both changes in run-time are inconsequential; overall performance is essentially identical. The larger increase in GC time with `_209_db` is primarily a result of incurring extra GCs (on average, 7 for Assertions compared to 6.1 for NoAssertions) because of the increased memory pressure from metadata on ownership assertions.

3.2 Qualitative evaluation

In addition to the performance benchmarks above, we tested our GC assertions on real-world code to search for memory leaks and other errors. We instrumented SPEC JBB2000 and `lusearch` from the Dacapo suite. In addition, we attempted to answer a question from the Sun Developer Network by instrumenting the attached program. We found that in most cases, GC assertions helped us find and repair problems quickly and precisely. In addition, GC assertions gave us a better understanding of how these programs worked.

3.2.1 SPEC JBB2000

SPEC JBB2000 is a benchmark that emulates a three-tier business system, with data stored in b-trees rather than an external database. Notably, it uses the factory pattern to create and dispose of objects. We first 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 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 path example in 1 shows the debugging output given by GC assertions when this error is detected.

The second problem we found was more subtle. In the main loop of the benchmark, the Company object from the previous iteration is destroyed (triggering a call to assert-dead) 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. Thus 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. Note that the object referenced by the `oldCompany` variable will be reclaimed on the following iteration when it is replaced by the Company that was allocated in this iteration. This is not a memory leak but an example of memory drag, where the Company object could be reclaimed earlier than it is. The Company data structure keeps a great deal of data live, and reclaiming it earlier reduces memory pressure on the system. Notice also that this problem could have been found by using assert-instances on the Company type, since there can only be one Company live in the benchmark at any given time.

Third, we investigated a known memory leak in SPEC JBB2000 first reported by Jump and McKinley [27]. 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 assert-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.

Finally, we revisited the issue of "dead" Order objects being reachable from Customer objects. One flaw of the assert-dead assertion is that the user must know where to place the assertions, i.e. where objects become unreachable. In SPEC JBB2000, we are lucky that the program includes destructors to deallocate objects, but in the general case Java programs will not have such information. Instead, we applied the assert-ownedBy assertion to the Orders in SPEC JBB2000. Orders are stored in an `orderTable` in each District, so we instrumented the `District.addOrder()` method and asserted that each Order added was owned by its `orderTable`. We found the same problem as before: Customer objects were keeping Order objects live after they were removed from the `orderTable`. The ownership assertion is an easier way to detect such problems since the user does not need to know when an object should be dead.

3.2.2 lusearch

`lusearch` is a benchmark in the Dacapo suite that tests the Apache Lucene text search engine library [19]. `lusearch` reads a pre-built index on disk and performs searches over the index using multiple threads.

This benchmark uses the Lucene IndexSearcher class to perform the searches. The Lucene documentation states, "For performance reasons it is recommended to open only one IndexSearcher and use it for all of your searches." [21] We instrumented `lusearch` with an assert-instances assertion stating that only one instance of IndexSearcher should be live. We found that for most of the benchmark's execution, 32 instances of IndexSearcher were live, one for each thread performing searches. This could be repaired by using only one instance of IndexSearcher and sharing it among the threads.

In this example, the programmer using the Lucene library was not aware of this performance recommendation. The library code could include an assert-instances assertion to warn a user if he tries to use more than one IndexSearcher instance in his code.

3.2.3 SwapLeak

We investigated a memory leak reported by Bond and McKinley [8]. The memory leak comes from a Sun Developer Network post where a user was asking for help understanding why an attached program runs out of memory [17]. The program defines a class SObject with a non-static inner class Rep with an instance field that points to a Rep instance. The SObject class defines a `swap()` method that takes another SObject and swaps the Rep fields of each.

The main loop of the program allocates a fixed number of SObjects and adds them to an array. It then iterates over the array, allocating new SObjects and swapping their Rep fields with those of the SObjects already in the array. The user expected that these new SObjects would be reclaimed after the swap, since they were not referenced by any local variables.

We instrumented the user's code with assert-dead assertions after the swap operation, and on execution we received the following warning:

```
Warning: an object that was asserted dead is
reachable.
Type: LSObject;
Path to object:  LSArray; ->
  [LSObject; ->
  LSObject; ->
  LSObject$Rep; ->
  LSObject;
```

This warning explains the problem. An SObject in the array has a reference to an instance of the Rep inner class, but that Rep instance maintains a pointer to a different SObject, one that we expected to be unreachable. The problem stems from the fact that non-static inner classes have access to other members of the enclosing class. Thus they must maintain a hidden reference to the enclosing class instance in which they were instantiated. Our GC assertions system displays this hidden reference and explains why the SObject instances were not being reclaimed.

4. Related Work

Our work is related to a variety of techniques for checking heap-based data structures and for detecting memory errors. These techniques can be roughly categorized according to

(a) how the desired properties are specified (ranging from programmer-written invariants to statistical analyses, such as anomaly detection), and (b) when and how often the checks are performed (either at compile-time, or at various granularities during execution.) GC assertions represent a particular point in this space: on the one hand, they require extra work to add to code, and there is no guarantee of when they will be checked; on the other hand, they provide the programmer with an expressive range checks and high-quality results, while maintaining extremely low run-time cost.

4.1 Runtime checking

GC assertions are closely related to program *invariants*, but differ in the balance between the strength of the guarantees provided and the performance of checking. Modeling languages, such as JML [12] and Spec# [2], allow programmers to add invariant specifications into their code, which are checked automatically at run-time. These systems ensure that the invariants always hold by checking them at every program point where they could be violated (for example, after every routine that updates a data structure). This approach, while complete, is extremely expensive – it can cause programs to run 10 to 100 times slower. Our system, on the other hand, checks heap properties very efficiently, but at essentially random program points (GCs). GC assertions, therefore, cannot technically be considered "invariants", since we can miss transient violations (those that do not persist across a GC boundary).

Recent work has used *incrementalization* to speed up runtime invariant checking by eliminating recomputation of the invariant check on parts of the data structure that have not changed [38, 25]. This technique is complementary to GC assertions: if we know that parts of a data structure have not changed since the last GC, we could avoid checking assertions for those objects.

HeapMD [13] monitors properties of objects (such as in-degree and out-degree) at run time and reports statistical anomalies as possible errors. ShapeUp monitors similar properties for Java, but uses type information to make checks more precise [28]. ShapeUp computes a *class-field summary graph* and reports anomalies in the in-degree and out-degree of its nodes. 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.

The QVM [1] Java virtual machine provides *heap probes*, which can be used to check some of the same properties as GC assertions and are also implemented using garbage collector infrastructure. The semantics of heap probes, however, are substantially different from GC assertions. Heap probes are performed immediately at the point the probe is requested. QVM triggers a garbage collection for *each heap probe* that must be checked, incurring a hefty overhead that is mitigated by sampling the heap probes rather than checking every single one. Our system, on the other hand, batches assertions together and checks them all in a single heap traversal during a regularly scheduled collection. As a result, checking is much more efficient, but it cannot verify properties at the exact point the assertion is made.

4.2 Static analysis

Previous work on static analysis has yielded a significant body of sophisticated techniques for modeling the heap and analyzing data structures at compile-time. Previous approaches include pointer analysis [11, 30, 5], shape analysis [24, 36, 26], type systems [23, 15, 9], and formal verification [31, 16, 42]. The strength of static analysis is that it explores all possible paths through the program: a sound analysis algorithm can prove the absence of errors, or even verify the full correctness of a data structure implementation. Static analyses, however, face three substantial challenges: (1) conservative assumptions about input values and control flow can lead to many spurious errors (false positives), (2) algorithms for building a detailed heap model scale poorly to whole-program properties, and (3) analysis typically fails for programs that use dynamic class loading, reflection, or bytecode rewriting. Our system builds on this work by supporting the *kinds* of data structure checks that have proved useful in static analysis, but avoids the pitfalls by checking them at run-time.

4.3 Instrumenting and controlling the JVM

Sun provides the JVM Tool Interface (JVMTI) [29] to allow tool developers to monitor runtime and GC activity. Several of our GC assertions could be implemented using JVMTI, with the advantage that they would be portable across different JVMs. We chose not to use JVMTI for three reasons. First, many of the hooks we need for GC assertions are optional parts of the specification. Second, JVMTI would not allow us to explore certain reporting mechanisms, such as the full object path. Finally, modifying the virtual machine incurs a lower performance overhead since we can perform low-level optimizations like using spare bits in object headers and changing the order of object traversal in the GC to speed up assertion checking.

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 [34] when an object is traced. Simplifiers provide a general mechanism for injecting code into the GC process, but the focus is primarily on improving program performance. Some of our GC assertions could be implemented using simplifiers: for example, `assert-dead()` could use a simplifier to check a flag in the object or object header. It would not be possible, however, to implement an assertion like `assert-ownedBy()`, which requires changing the order of traversal of the object graph.

The COLA system allows programmers to dictate the layout order of objects to the garbage collector using an

iterator-style interface [33]. Like simplifiers, the focus of COLA is on controlling the garbage collector's behavior to improve performance.

5. Conclusion

The garbage collector is a powerful source of information about large-scale program state and behavior because it systematically visits all objects and references in the heap. It is in a unique position to check a wide variety of data structure properties. Furthermore, the garbage collector can check properties, such as object lifetime, that no other subsystem has access to. This paper presents a programmer-driven technique for taking advantage of these capabilities by providing a structured way to communicate with the garbage collector. GC assertions are easy to use and provide accurate results with high-quality debugging information. By piggybacking assertion checks on the existing GC tracing algorithm, GC assertions are cheap enough to be used in deployed software, where they can help detect the most important and serious bugs: those that occur during real executions.

Acknowledgments

We would like to thank Mike Bond, Kathryn McKinley, Nick Mitchell, Nathan Ricci, Gary Sevitsky, Yannis Smaragdakis, and Ben Wiedermann for their helpful ideas and discussions. We thank Steve Blackburn for his help with the measurement methodology. We thank the anonymous reviewers for their helpful comments. Finally, we thank the Jikes RVM team for their great work in building an important platform for research.

References

[1] M. Arnold, M. Vechev, and E. Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 143–162, New York, NY, USA, 2008. ACM.

[2] M. Barnett, K. Rustan, M. Leino, and W. Schulte. The spec# programming system: An overview. http://research.-microsoft.com/users/leino/papers/krml136.pdf.

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

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, 2002.

[5] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *ACM Conference on Programming Languages Design and Implementation*, pages 103–114, New York, NY, USA, 2003. ACM.

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

[7] 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.

[8] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 109–126, 2008.

[9] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symposium on the Principles of Programming Languages*, pages 213–223, 2003.

[10] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 441–460, 2007.

[11] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Languages Design and Implementation*, pages 296–310, 1990.

[12] Y. Cheon and G. T. Leavens. A runtime assertion checker for the java modeling language (jml). Technical Report TR 03-09, Iowa State University, 2003.

[13] 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.

[14] 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.

[15] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *SIGPLAN Notices*, 33(10):48–64, 1998.

[16] P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 363–382, 2006.

[17] S. D. N. Forum. Java programming [archive] - garbage collection dilema (sic). http://forums.sun.com/thread.jspa?-threadID=446934.

[18] A. S. Foundation. Apache http server project. http://httpd.-apache.org/.

[19] A. S. Foundation. Apache lucene - overview. http://lucene.-apache.org/java/docs/index.html.

[20] A. S. Foundation. Apache portable runtime project. http://-apr.apache.org/.

[21] A. S. Foundation. Indexsearcher (lucene 1.9.1 api). http://-lucene.apache.org/java/1_9_1/api/org/apache/lucene/search/-IndexSearcher.html.

[22] J. Fox. When is a singleton not a singleton? http://java.sun.-com/developer/technicalArticles/Programming/singletons/.

[23] P. Fradet and D. L. Métayer. Shape types. In *ACM Symposium on the Principles of Programming Languages*, pages 27–39,

1997.

[24] 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 *ACM Symposium on the Principles of Programming Languages*, pages 1–15, 1996.

[25] M. Gorbovitski, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient runtime invariant checking: a framework and case study. In *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*, pages 43–49, 2008.

[26] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *ACM Symposium on the Principles of Programming Languages*, pages 310–323, 2005.

[27] 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.

[28] M. Jump and K. S. McKinley. Dynamic shape analysis. In *ACM International Symposium on Memory Management*, 2009.

[29] Jvm tool interface. http://java.sun.com/javase/6/docs/-platform/jvmti/jvmti.html.

[30] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM Conference on Programming Languages Design and Implementation*, pages 56–67, 1993.

[31] S. McPeak and G. Necula. Data structure specifications via local equality axioms. In *Computer Aided Verification*, pages 476–490, 2005.

[32] 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.

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

[34] 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.

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

[36] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symposium on the Principles of Programming Languages*, pages 105–118, 1999.

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

[38] A. Shankar and R. Bodík. Ditto: automatic incrementalization of data structure invariant checks (in java). In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 310–319, New York, NY, USA, 2007. ACM.

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

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

[41] X. Wang, Z. Xu, X. Liu, Z. Guo, X. Wang, and Z. Zhang. Conditional correlation analysis for safe region-based memory management. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 45–55, New York, NY, USA, 2008. ACM.

[42] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *ACM Conference on Programming Languages Design and Implementation*, pages 349–361, 2008.