GC Assertions: Using the Garbage Collector to check heap properties

Shirley Gracelyn

February 16, 2011

1 Motivation

Automatic memory management has definitely relieved the burden from the programmers. But, it also does not provide any information about the memory behavior of the program to the programmer. This is a problem, as memory which is reachable , but which might never be used again will not be reclaimed. Since the programmer might know the points in the program, where an object will become unused, the hints he provide might be useful to debug the program to make sure there are no obsolete references.

For example, some of the questions the programmer might have about the memory itself will be as follows: Is there only a single instance of a given type? Will this object be reclaimed at the end of next garbage collection cycle?Are there any outstanding references to this object?These are questions which the Garbage Collector is in a unique position to provide an answer for, as during GC tracing, every object in the paper will be looked at.

2 Main idea

This paper introduces a set of GC assertions, which can be checked for at different Garbage Collection cycles of the program. These assertions differ from the normal assertions, mainly because the GC assertions are not checked immediately. Rather, they just indicate a condition should be checked, and this check is piggybacked into the GC cycle.

The two main goals of this paper are that the set of assertions should be expressive to check for different memory anomalies and also the run time overhead of these assertions should be kept low. To keep the run time overhead low, is an important requirement. Consider a situation where a programmer inserts a lot of GC assertions in the code. What he definitely might not want is to slow down the performance of the program, just to check these assertions.

Some features available with GC assertions are as follows: These checks are done at run time, and is verified on top of the actual concrete heap. Hence there are no assumptions made due to the dynamic features of the language, which might not always be true and can also give misdirecting information about the heap state. Any report given, will have to be checked for manually in case of static analysis or heuristic checking to ensure that it indeed is a problem. but in case of assertions, the checks are done and the required debugging information is also provided to the programmer, so that he can figure out the nature of the anomaly. These assertions give accurate information and are application specific, since the assertions are inserted at particular points in the program depending on the expected life time of an object in that specific application. Also, since the checks are pushed until garbage collection time, these assertions does not impose added overhead.

3 Implementation

The GC assertions for this paper are implemented in Jikes RVM 3.0.0. The garbage collector used is Mark and Sweep collector, as it will be checking the entire heap at each cycle. This is important in case of GC assertion checking, as we will want to ensure that the assertion is preserved as early as possible. In case of incremental algorithms, there might be problems because, some objects might not be looked at for a longer time, hence some errors might go undetected. Modifications are done to the GC algorithm, such that with minimal space and time overhead, the GC assertions can be checked for.

4 Lifetime assertions

These check for the life time properties of the assertions, by marking the appropriate bits of the objects when the assertion is encountered, and later checking for these marked objects during garbage collection.

4.1 assert-dead(p)

This assertion is triggered when the object pointed to by p is not yet reclaimed during garbage collection. This assertion is mainly used to check, if at a given point in a program , an object which might never be used again is still reachable. This assertion is implemented by setting a spare bit in the object's header pointed to by p. In the next garbage collection cycle, if there are any objects with the assert-dead bit is set, a warning is thrown.

No space overhead, as only spare bits in the objects are used for this assertion. Time overhead just involves checking for the appropriate bit in the objects, which does not impose any added penalty as the object's header will have to be checked anyway during the garbage collection process. This provides the main advantage of adding the assertion checking to the GC process itself, so that the assertions can be verified for, with no slow down in performance.

4.2 assert-alldead()

This is used in conjunction with a start-region() assertion. The start-region assertion marks the region of the code, in which any object allocated will be checked for reachability during garbage collection. This will be used to check if there are any memory leaks from one section of the code to the other.

This is implemented with help of a bit specific for each thread, that just specifies if the current code region executed by the thread is in scope of a start-region assertion. Any allocation done when this bit is set, adds the object to the queue. On a call to assert-dead, the region bit is reset, and assert-dead is called on each object in the queue.

Space overhead for this assertion depends on whether the region is currently active. If inactive, each region has a thread specific flag bit and queue reference. If active, it has an additional word for each object that was allocated after the start-region assertion, which can be reclaimed when the region becomes inactive.Time overhead will be to check the region bit and to add the object in the queue.Since assert-dead call is reused, there is no extra time overhead here.

5 Volume assertions

These are type specific assertions, which expresses the requirement of how many instances should exist for a given type. These assertions can be checked while scanning the heap, and verifying the number of instances against the specified constraint.

5.1 assert-instances(T,I)

This assertion helps in two different goals. To ensure that the application is following a singleton pattern might be a difficult thing to check for correctly. Reasons are that, if there is a sub class which extends from a class and the super class has a public constructor, then anyone can make instances of the sub class.Because a subclass is essentially an instance of the super class, there will be now multiple instances of the singleton, as the uniqueness constraint cannot be imposed at compile time without a private constructor.

Another use will be for performance reasons, as sometimes an object of a certain type should be limited for performance reasons.

Since this is a condition associated with type, the RVMClass is itself be modified to maintain the instance limit and count for the class. These are set when the assert-instance call is encountered, and during GC every time an object of a type is seen, the count is incremented.Space overhead here will be to maintain two words per loaded class, and also one word per tracked type in an array of tracked types. Time overhead is minimal.

6 Ownership assertions

These help the programmers to check for the connectivity between the objects and data structures.

6.1 assert-unshared(p)

This assertion is triggered if the object pointed to by p has more than one incoming pointer. It is used in several cases to check that a property of a data structure itself is not violated - one example, as cited in the paper is to check if a tree data struucture has not been suddenly changed to a DAG or a graph. A bit is set to indicate that the object should not be shared, and this will be checked for during garbage collection.

6.2 assert-ownedby(p,q)

This assertion will be triggered if object pointed to by q is not owned by object pointed to by p. This requires a clear definition of owner and ownee. One of the definitions is that, all paths from roots through heap must pass through the owner. But this will be too restrictive to be practical, as there are common constructs like iterators which violate this definition.

This paper defines ownership as follows: The set of paths through heap to the ownee must include at least one path that passes through the owner. This allows other objects to reference the ownee, but there should be at least one path to the ownee reachable from the owner. This definition makes sense, as an object should never outlive its owner. Also, this is easier to check for, as rather than figuring out the point where an object should be dead, the programmer can identify the owner of the object.

To implement this assertion, the owner/ownee pairs should be checked for in a single garbage collection. A simple implementation will be to check if the owner is along the path, when an ownee is seen during GC tracing. But, if the owner is followed by a previously marked object, repeating the tracing information to find if the ownee is reachable from the previously marked object. One alternative was to bubble up the ownee information along the path, which will be prohibitive in terms of space and time overhead.

The strategy proposed by the authors to handle this issue is to start GC tracing with the owner objects, assuming the owners themselves are alive, which might not be the case always. So, in the first pass, a scan is done beginning from the owners, and an ownee object can be verified for, to check if it belongs to the current owner, else a warning is issued. The owner itself is not marked in this phase. In the second pass, the normal scan starting from the roots is done, and an ownee object that is not marked as owned will trigger a warning.

If the owner itself is not reachable during the second phase, this object will be garbage collected. But any objects which were previously reachable through the owner will not be reclaimed in this GC cycle. This might cause a memory drag issue.

Another important problem which could happen is when data structures overlap. In this case, depending on which path might have been chosen, the ownee might have already been marked when the path from the owner is traced. But it will not be set for the right owner, and hence might throw a false positive when a scan is done from the owner. Easy solution for this problem will be to just constrain the data structures to be disjoint, but this will not be true in case of real programs, as the objects usually have back edges, causing overlap. In the implementation, the scan is stopped when an owner is reached, solving the back edge issue. Only thing enforced is that regions of the heap belonging to different owners i.e., their paths from the owner to the owned objects itself should not overlap, as then , as described above, a false warning might be issued unnecessarily.

7 On violation of assertions

1. Log an error, but continue executing
2. Log an error and halt
3. Force the assertion to be true

7.1 Debugging information

The generated report provides the full path through object graph from root to dead object. Only limitation here will be that, we might not be able to identify the offending object in all cases. For example, in assert-unshared call, the second path will inform that there is a violation, but printing only that might not be helpful to the programmer, if the problem exists in the first path.

8 Evaluation

Example 1: Consider an order processing system, where destroy() is called on an Order object. But, assert-dead assertion on Order objects failed because the Customer objects still had a last order field holding the reference to Order object.To solve this problem, the back pointer from the Order object is explicitly set to null.

Sample output might be as follows:

```
Path: Lspec/jbb/Customer; ->
[Ljava/lang/Object; ->
Lspec/jbb/LastOrder; ->
[Ljava/lang/Object; ->
Lspec/jbb/Order;
```

Example 2: Local variable retains a reference to Company object from the previous iteration.This results in a memory drag, as the data structure will be reclaimed in next iteration.This violation can be detected through calls to assert-instances, as more than one Company instance should not be alive at the same time.

9 Performance

The slow down in performance is tested against two different types of configuration here - Base configuration(no change in platform and no assertion checks), Infrastructure configuration(Modified platform but no assertion checks). The infrastructure configuration itself does not have a notable increase in run time and GC time, which is an optimistic behavior, as if programs are run with no assertions on the modified platform, it does not cause any overhead.

With assertions, the run time itself does not increase much as compared to the base and infrastructure configurations, proving that the idea to add the checks to the garbage collection time is a fruitful idea. But, since there are more checks involved during garbage collection time, GC time increased by around half. This is expected, and is not a huge cost to be paid for checks on very large number of objects.

10 Limitations

To put assert-dead assertion at the right place, programmer should know when the object should be dead, i.e where the objects become unreachable. An easier and stable way to detect the error, is to insert an ownership assertion, which ensures that an exception will be thrown when an object outlives its owner.

The set of assertions is fixed, and cannot be customized. This limits the expressiveness of the programmer. Another important issue which will be faced by the concept of GC assertions, is that since the assertions are checked only during the garbage collection cycle, there might be incorrect evaluation due to some changes in the heap, depending on how long after the assertion call, garbage collection happens.