Bounding Space Usage of Conservative Garbage Collectors

Hans-J. Boehm*
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
Hans_Boehm@hp.com

ABSTRACT

Conservative garbage collectors can automatically reclaim unused memory in the absence of precise pointer location information. If a location can possibly contain a pointer, it is treated by the collector as though it contained a pointer. Although it is commonly assumed that this can lead to unbounded space use due to misidentified pointers, such extreme space use is rarely observed in practice, and then generally only if the number of misidentified pointers is itself unbounded.

We show that if the program manipulates only data structures satisfying a simple *GC-robustness* criterion, then a bounded number of misidentified pointers can result at most in increasing space usage by a constant factor. We argue that nearly all common data structures are already GC-robust, and it is typically easy to identify and replace those that are not. Thus it becomes feasible to prove space bounds on programs collected by mildly conservative garbage collectors, such as the one in [2]. The worst-case space overhead introduced by such mild conservatism is comparable to the worst-case fragmentation overhead for inherent in any non-moving storage allocator.

The same GC-robustness criterion also ensures the absence of temporary space leaks of the kind discussed in [13] for generational garbage collectors.

1. INTRODUCTION

Garbage collection, or automatic memory reclamation, allows unreachable memory objects to be automatically recycled, providing two kinds of benefits to the programmer:

Convenience The programmer does not need to provide code to explicitly deallocate memory. Nor do interface specification need to deal with issues of object "ownership" to assign deallocation responsibility. This can substantially simplify both interface specifications and programs that use them. Less frequently, it can result in significant application speedups.[1]

Safety Since the programmer does not explicitly deallocate memory, there is no danger of accidentally reusing an object's memory while the object can still be accessed. This prevents one module from corrupting the data structures of another module to which it should not have access, greatly simplifying fault diagnosis, especially in large projects. It also makes it possible for the rest of the language implementation to guarantee interesting safety and security problems, *e.g.* that a module cannot read or write arbitrary sections of memory by misinterpreting an integer as a pointer.

In this paper, we will be interested in tracing garbage collectors, which are most commonly used in language run-time systems. These traverse all reachable memory objects, and then make the remainder of the heap available for reallocation to new objects. Traditionally this has required that the garbage collector be able to precisely identify all pointer variables (*roots*) and be able to accurately locate all pointer fields inside heap objects. Such collectors are commonly described as *type-accurate*.

Conservative garbage collectors have relaxed this requirement to tolerate *ambiguous* pointers, i.e. locations which may or may not contain pointers. Ambiguous pointers whose value is a valid object address are treated as though they were pointers, in that the data structure they reference may not be reclaimed.

They have several advantages leading to their continued use, *e.g.* in some of IBM's Java virtual machines, in the runtime for the GNU Java compiler, in Amazon.com's web server, in some Xerox printers, in many Scheme implementations, etc.

In particular:

1. They can be used with C/C++ programs and the vast majority of preexisting binary libraries. The C/C++ compiler must obey some additional correctness constraints in that pointers must remain recognizable by the collector. Empirically standard compilers satisfy this constraint for all programs at low optimization levels, and for nearly all programs at high optimization levels. This typically requires a large degree of

*Some of this work was done while the author was employed by SGI.

[1] See for example **http://www.hpl.hp.com/personal/Hans_Boehm/gc/example.html**

conservatism, in that essentially all pointers will be ambiguous.

2. They can be used with compilers that generate C code. This typically requires less conservatism in the collector, since the compiler may be able to describe pointer locations in heap objects.

3. They allow the collector/compiler interface to be much simpler. In the case of a fully conservative collector for C/C++ it is almost nonexistent. In other cases, the collector may elect not to describe pointer locations in a few cases in which it would be difficult or expensive to do so, *e.g.* for the top frames of thread stacks [2]. Many *type accurate*, i.e. non-conservative collectors introduce time or space overhead into non-allocating code as part of the collector interface; this is much less of an issue for conservative collectors.

4. They facilitate language interoperability. They allow multiple language implementations to more easily share a single garbage collected heap, since there needs to be far less agreement on the collector interface [15]. In the case of a mixed Java/C program, they allow the collector to scan the C variables for pointers to Java heap objects, thus eliminating the need for complex, relatively error-prone, and sometimes expensive interfaces such as JNI [10].[2]

5. Since language interoperability is easier, it also tends to be easier to write C language run-time support code for something like a Java virtual machine. That support code is likely to be more efficient, since it does not need to worry about explicit communication with the garbage collector. Small degrees of conservatism provide most of the advantage here; it often simplifies matters greatly if the collector can simply see variables local to the run-time system on the stack.

On the other hand, conservative collectors have the disadvantage that they restrict object movement. There is no way to safely update a "pointer" to a moved object, unless we are certain that it was really a pointer.[3] Although it is often a performance advantage not to move objects, it appears to be a significant performance disadvantage for programs that allocate primarily large numbers of very short-lived objects.

Conservative collectors also risk retaining unreferenced memory as a result of misidentified pointers. A value v of some other data type, *e.g.* an integer, may be misidentified as a pointer to a large, and possibly growing, data structure, which is not really reachable. In this way v may cause the collector to retain large amounts of unreachable memory.

Empirically, this occasionally has some effect on space usage, but this affect is rarely catastrophic[8, 4]. The large majority of applications behave perfectly well with a conservative garbage collector. A few applications don't, especially if the collector is too naive. But in those cases, the failure is generally easily detectable during testing, and can be avoided by communicating small amounts of type information to the garbage collector.

[2]Note that the reference is a 300 page book on JNI, described by the amazon.com reviewer as "densely written". It is not the only book-length treatment of JNI.

[3]It is still possible to move objects referenced only by unambiguous pointers, c.f. [3].

We are only aware of one explicitly deallocating program in which replacement of explicit deallocation with conservative garbage collection failed unavoidably. And in that case conservative pointer tracing was not the issue; the program relied on deallocating reachable, but unaccessed memory. Thus no garbage collector could have solved the problem.

But our concern here is not the empirical or typical performance of conservative garbage collection. That issue is addressed well by [8]. We would instead like to address the concern that a conservatively collected program may unexpectedly retain large amounts of memory due to a particularly unfortunate misidentified pointer, which did not arise during testing. We would like to bound space usage independently of the particular pointer-like values that arise during a particular execution.

We do this both be proving a mathematical bound on space consumption, and by suggesting a testing technique for identifying *potential* unbounded growth without actually needing to find an execution for which a particularly unfortunate pointer misidentification actually occurs. The primary goals are to

1. Get us closer to making conservative garbage collectors safe for environments that require hard space bounds, such as embedded environments,

2. To provide a more intuitive explanation of why conservative collectors in fact behave reasonably well in practice, and

3. To provide a better characterization of what can provoke failure of conservative collectors, so that it can be avoided.

Our results can also be used to bound the damage caused by an inappropriate object promotion in a type-accurate generational garbage collector.

1.1 An Embarrassing Failure Scenario

Some data structures do not interact well with conservative garbage collection. Lazily evaluated infinite lists are one such example. Fortunately for us, these are rarely used with conservative garbage collectors.

A more common example, mentioned in [16, 4], is a simple queue data structure, implemented as a singly linked list, with a head and tail pointer. Elements are inserted into the tail position and removed at the head by advancing a head pointer. Removed elements should be reclaimable by the garbage collector. But a single false pointer to one of the queue elements will prevent any further such reclamation, as in figure 1. Thus if the active queue length is intended to be bounded, a conservative collector may arbitrarily increase space consumption as the result of a single false reference.

We make several observations about this example:

1. This is an unpleasant situation, given our goal. The presence of such data structures clearly precludes proving any interesting bounds on space consumption.

2. It is not immediately apparent how to test for this situation. We ran a simple toy program based on this data structure on Linux/X86 with our fully conservative collector[4], and no apparent space leaks. Unfortunately, a single misidentified dynamically-created

[4]cf. **http:/www.hpl.hp.com/personal/Hans_Boehm /gc**

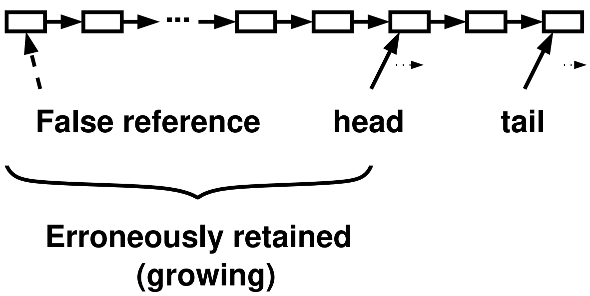

Figure 1: False reference retaining unbounded space in a queue

"pointer" could introduce such a leak, and indeed it did appear with more collector debugging facilities enabled.[5]

3. The above data structure is easily fixable. Clearing the `next` field in the list element being removed from the queue will avoid the potential disaster. This kind of "fix" is still much safer than resorting to manual storage management, since errors cannot corrupt unrelated data structures. It negates the convenience advantage of using a garbage collector, but it preserves the safety advantages.

4. The problem is not limited to what is normally referred to as conservative garbage collection. It could also be caused by having the garbage collector accidentally trace from a dead compiler-generated variable, or by promoting a queue element to a generation that is effectively not collected.[6] This data structure is safe only with a system that is carefully designed to prevent any form of extra space retention (cf. [1]).

The next section is concerned with "well-behaved" data structures that bound the damage caused by spurious pointers, or by accidental promotion in a generational collector. We argue that most commonly encountered data structures are well-behaved in this sense.

2. BACKWARD-FORWARD REACHABILITY

In the following, and in the rest of the paper, we say that an object is *reachable*, *conventionally reachable*, or *properly reachable* from another other object if it can be reached by following real, not misidentified, pointers form the other object. If no other object is mentioned, we mean that it is accessible by following real pointers from a program variable or *root*. The *reachable* objects are precisely those that a type-accurate, i.e. non-conservative collector would retain.

[5]The "black-listing" technique of [4] will usually prevent this from happening as the result of a misidentified preexisting constant value. The value would have to be generated after or near the time the corresponding memory object is allocated, and then remain constant.

[6]For example, VisualWorks Smalltalk apparently has a permanent object space, to which all old objects can be moved as a result of an explicit client call, but which is not implicitly collected. An accidental promotion of a single queue element to this space would have the same effect. Similar problems can occur in more conventional generational collectors, but the damage would be temporary, since all generations are eventually collected. Nonetheless it could cause appreciable performance degradation.

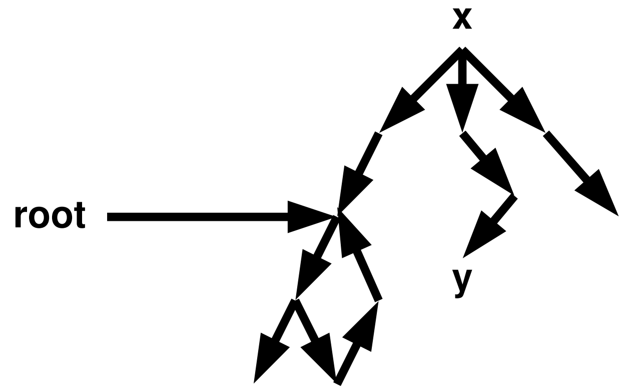

Figure 2: Objects backward-forward-reachable from root through x.

OBSERVATION 2.1. *Unreachable objects are never modified. In particular objects erroneously retained by a conservative collector cannot be modified after they become unreachable. We rely heavily on this rather obvious fact.*

On object x is a predecessor of y if x points directly to y. We continue to use this terminology to refer to the original graph even in contexts in which we also need to discuss the reverse of the original edges.

We define the set of objects that are *backward-reachable* from an object x to be those objects y such that x is reachable from y. Define the set of objects backward-reachable from a root pointer to be those objects backward-reachable from the object to which it refers. An object is backward-reachable if it is backward-reachable from any root.

An object y is *backward-forward-reachable* from a root r through an *anchor* x iff x is backward-reachable from r and y is reachable from x. (See figure 2.) An object y is *backward-forward-reachable* if there is an object x such that y is backward-forward reachable through x. An object is backward-forward-reachable if it is backward-forward-reachable from any root.

Call a data structure, i.e. a set of objects, strongly connected if, for any two objects x and y in the set, y is reachable from x if and only if x is reachable from y, i.e. an object is reachable from another if and only if it is backward-reachable. Data structures such as doubly linked lists, or trees with parent pointers that maintain back-pointers corresponding to all forward pointers, are particular instances of this.

THEOREM 2.1. *If a program only builds strongly connected data structures, that is if x is reachable from y whenever y is reachable from x, then the set of backward-forward reachable objects is the set of reachable objects.*

Proof Trivial. •

THEOREM 2.2. *If a program is purely "functional", in the sense that an object's fields are set only during initialization, then the set of all objects S backward-forward reachable from a root r through an anchor x, was at some point reachable from a single root.*

Proof The object x must have been accessible through a root at some point. Since objects are not changed, all of S was reachable through x at that point. •

We define a program to be *strongly GC-robust* if it satisfies the conclusion of the preceding theorem, that is if all objects backward-forward reachable from a root through a single anchor were at some point reachable from a root. We say that a data structure is strongly GC-robust if this statement applies when restricted to objects that are part of that data structure. Both data structures created by functional programs and circular data structures are strongly GC-robust.

Note that we are really applying this definition to the garbage collector's view of the data. In particular, although lazy functional programs can be reasoned about as though they did not update objects, the implementation, as seen by the garbage collector, does. Thus we do not consider them "functional" in the above sense. Lazy functional programs manipulating infinite data structures are generally not GC-robust, leading to the issues observed in [16, 13].

3. A SPECIFIC BOUND

In the following we assume:

1. A garbage collector will only consider memory objects to be reachable if they were properly allocated in the past, and thus were at some point reachable. Pointers to unallocated memory are never considered valid.
2. An object that was last modified by the collector will not subsequently be viewed as live by the collector, before having been reallocated. Effectively this means that once the collector has reclaimed an object, it will stay reclaimed until it is reallocated. This may take some care if the collector builds free lists, which might be referenced by false pointers.
3. Objects that appear to be reachable to the collector can be modified only by the client (mutator).
4. Any modified object is directly pointed to by a root at the time of modification.

We expect that these hold for all common conservative collector implementations.

THEOREM 3.1. *Consider a conservative garbage collector which misinterprets a single arbitrary value v as pointer. The set of all objects S that appear reachable from v, i.e. that are reachable from the object x containing address v, were at some point backward-forward reachable from a single root through a single anchor. The anchor can be chosen to be x.*

Proof Let y be the last object in S to be created or modified. Let time t be immediately after y's creation or modification. Clearly at that point y was pointed to by a root r. Since this is the last modification of S, y must already have been reachable from x at this point. Thus x was backward reachable from y, and thus all of S was backward-forward reachable from r through anchor x at time t. •

COROLLARY 3.2. *If a program uses only strongly GC-robust data structures, and the number of pointers misidentified by a garbage collector is bounded by N (e.g. because only the stack is scanned conservatively, and its depth is bounded, or because only the top stack frame is scanned conservatively, or because we only conservatively scan stack frames corresponding to the run-time system), then the extra space returned by conservative pointer scanning is bounded by N times the maximal amount of live memory.*

Proof Follows trivially from the preceding theorem and the definition of strong GC-robustness. Each misidentified pointer retains a data structure that was backward-forward reachable, and hence reachable at some point during program execution. Thus the memory retained by a single misidentified pointer is bounded by the maximal amount of live memory used during program execution. •

The above corollary gives a rather extreme bound on excess memory retention, but a bound nonetheless. The bound itself may be sufficient to give hard guarantees in a few cases, *e.g.* if there are only a few conservatively scanned roots, and/or the program maintains many disconnected data structures, only one of which can be retained by a false reference. In any case, it helps to explain why real conservatively collected programs don't grow without bounds, and rarely retain significant amounts of extra memory.

4. TESTING FOR POTENTIAL LEAKS

The above gives a provable space bound on conservatively garbage-collected programs. We assumed that the underlying program was strongly GC-robust.

In many cases we are less concerned with a hard space bound than with some empirical assurance that the program will not exhibit unbounded space leaks. However, we also often do not have a proof that all the programs data structures are strongly GC-robust. We would like to be able to test such programs for potential space leaks.

As we saw in our motivating example, even non-GC-robust programs may run in apparently bounded space, simply because no false pointers happen to reference the offending data structure. Thus simply testing such programs as we would a non-garbage-collected program may not be satisfactory. Here we explore a method for explicitly testing a conservatively collected program for potentially unbounded space leaks due to non-GC-robust data structures.

4.1 Characterizing unbounded growth

In general, the memory reachable from a single misidentified pointer v can be pictured as in figure 3. The misidentified pointer v "points to" an object x from which we can reach a set U of otherwise unreachable objects. From x we may also be able to reach an additional set of objects R, which are also reachable from proper roots. Objects in R are still modifiable by the client, but objects in U are not.

LEMMA 4.1. *Assume that non-pointer value v is an address inside object x. At time t_1 let U_1 be the set of objects reachable from x, but not reachable from proper roots. Let U_2 be the corresponding set of objects erroneously retained by v at some later time t_2. Any objects in U_2 but not in U_1 must have become unreachable (from proper roots) between t_1 and t_2.*

Proof Objects in U_1 cannot change in the interim, since they are not actually accessible to client code. Thus the only way for the set of improperly retained objects to grow is for an object previously reachable from both x and a proper root to become unreachable. •

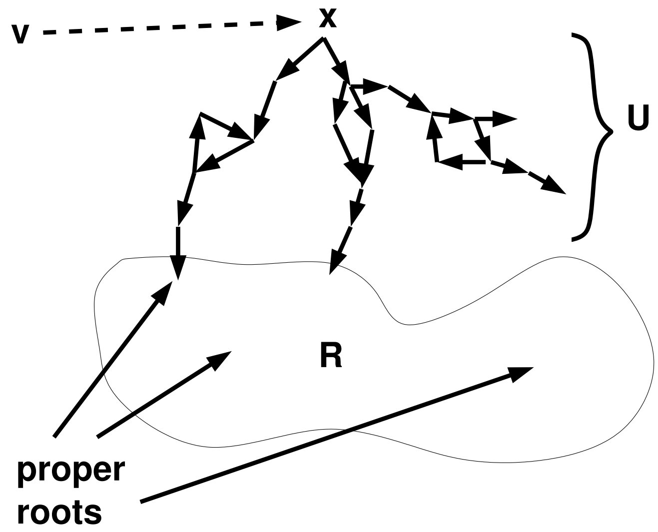

Figure 3: False reference pointing to reachable and unreachable objects

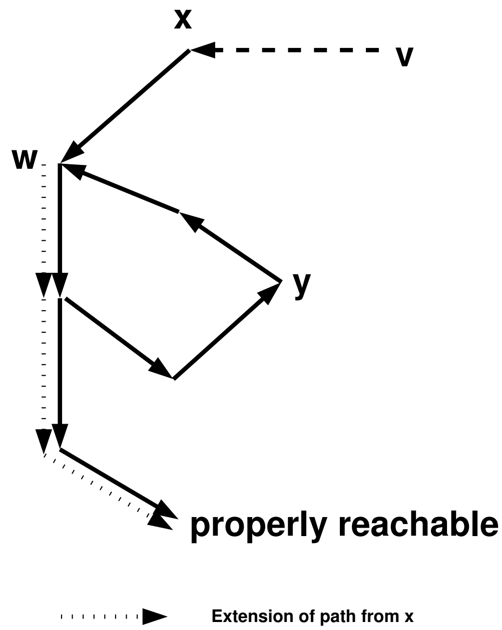

Figure 4: False reference pointing to reachable and unreachable objects

THEOREM 4.2. *Again assume that a non-pointer value v is an address inside object x. If the number of objects reachable from x but not reachable from proper roots grows without bounds, then the length of the longest simple path from x through unreachable objects to a reachable object also grows without bounds.*

Proof Let U be the set of root-unreachable objects reachable from x. By the preceding lemma, the set of objects erroneously retained by v can grow only if at least one object y accessible from x becomes inaccessible from proper roots. We consider two cases:

1. There is a path from y to a properly reachable object. In this case y must be on a path from x to a reachable object. If it is on a simple path, then that path must have grown longer as a result of y becoming unreachable. If it is not on a simple path between x and a reachable object, then consider a minimal path from x to a properly reachable object z through y, as in figure 4. Let w be the first object on this path to be repeated. The object w must be reachable from y. (If the cycle didn't contain y, the path wouldn't be minimal.) Thus the path from x to the previously reachable object w was extended to a longer path to a reachable object.

2. There is no path from y to a reachable object. In this case, no path from x to a reachable object is extended. But at any point there are only a fixed number of such objects y, and each one can be made inaccessible at most once. Thus this case can only occur a finite number of times between instances of case 1.

Thus we must grow a simple path from x to a reachable object infinitely often. Since the out-degree of each object is finite, this is would be impossible if the length of the longest such path where bounded. •

This implies that if the length of simple paths through unreachable objects ending in a reachable object remains bounded, then all paths from a possible false reference to a reachable object remain bounded, and hence the amount of unreachable memory that can be retained by a single false reference remained bounded. Thus this characterizes the potential for unbounded leaks without having to consider all possible potential false references.

OBSERVATION 4.1. *The above theorem still holds if we approximate the longest simple path by performing a depth-first-search of the backwards reachability graph, and simply delete the depth-first-search back-edges, i.e. the edges to objects currently on the depth-first-search stack. The above argument still applies to the remaining paths.*

This has the advantage that while finding the longest simple path in a directed graph is NP-complete (see [7], problem ND29), the same problem for the resulting acyclic graph is efficiently computable. We assign a *height* to each vertex during a depth-first search of the reverse reachability graph. The height is computed simply as one plus the maximum height of its predecessors (ignoring the previously mentioned edges introducing cycles). Since the predecessors of an unreachable node are also unreachable, this height must remain constant once it has been computed, and hence needs to be computed only once.

In our motivating queue example, the height of each discarded object is one more than that of its predecessor.

4.2 Weak GC-robustness

We define a program to be *weakly GC-robust* if the length of simple paths through unreachable objects ending at an object in the data structure remains bounded for any execution, or equivalently, if the height of objects pointed to by unreachable objects remains bounded. By the preceding theorem, weakly GC-robust programs cannot exhibit unbounded space leaks due to a single misidentified pointer.

By analogy to the earlier terminology, we call a data structure weakly GC-robust if simple paths through its no longer reachable objects remain bounded.

THEOREM 4.3. *Strongly GC-robust programs (or data structures) with a bounded number of reachable objects are weakly GC-robust.*

Proof Assume the program is not weakly GC-robust, but is strongly GC-robust. There is an unboundedly growing path starting at x, ending in a reachable object. For any object p in this path, which became unreachable at time t, consider the last time t' at which this object, or another object p', preceding it in the path, had just been allocated or modified. At time t', the object p' must have been referenced by some root r. Thus the entire path was backward-forward reachable from r through x at t'. Thus from the definition of strong GC-robustness, the entire path from x to p must have been reachable at some point. Since this is true for arbitrary p, and hence arbitrary path length from x to p, the number of reachable objects is not bounded. •

OBSERVATION 4.2. *The converse of the preceding theorem does not hold.*

Consider the queue data structure in our motivating example, but restricted so that only a bounded number of elements are ever removed. This structure is weakly GC-robust, but not strongly so. The strong GC-robustness criterion allows us to impose precise bounds on the cost of misidentification. Our notion of weak GC-robustness was designed to allow us to reason about bounded vs. unbounded space loss. Whether there are better ways to characterize these is an open issue.

CONJECTURE 4.1. *All common data structures, except the straight-forward implementations of singly-linked queues and infinite data structures relying on lazy evaluation, are weakly GC robust.*

4.3 A testing algorithm

We can now give a possible strategy for testing that an application will not grow without bounds as the result of a bounded number of pointer misidentifications. We modify the garbage collector so that:

1. Before each collection, we build an auxiliary data structure allowing us to quickly find all pointers to a particular object. Effectively, this is the backwards points-to graph.

2. As with the unmodified collector, we trace all objects reachable from normal roots, marking the objects we encounter.

3. Using the data structure from step 1, we perform a depth-first search of the backwards reachability graph, computing the height of both unreachable objects, and of reachable objects directly referenced by unreachable objects. We report the maximum height computed for a reachable object directly referenced by an unreachable one.

4. We arrange to retain the heights of any reachable objects pointed to by unreachable ones. This will be used as a starting point for future height computations. Since the unreachable part of the graph cannot change in the future, the remembered value will continue to describe its contribution to the height of the reachable object. We thus do not need to retain the unreachable part of the graph for future collections.

5. We discard unreachable objects as for a normal collection. We also discard the reverse reachability graph and any other auxiliary data structures, except for the height information from the preceding step.

If the reported maximum heights appear to remain bounded, we conclude (subject to the usual uncertainties of program testing) the the program is weakly GC-robust, and is thus not subject to unbounded pointer misidentification leaks. If the heights appear to grow without bound, we can identify the source of the growth, and hopefully repair it with explicit pointer clearing or the like.

4.4 Empirical observations

We built a prototype implementation of the above framework inside our garbage collector. It shares some of the underlying infrastructure with that used in [14], and could easily be made to report, for example, the allocation sites for the objects responsible for increasing heights. Thus it could be used to easily isolate non-GC-robust data structures, though we do not currently have a very attractive user interface.

Here are some resulting observations:

- Based on a small sample of test programs, of which Ghostscript[7] was the only nontrivial application, the maximum height of objects for well-behaved applications stabilizes quickly, and is usually on the order of at most tens or hundreds.

- In spite of the fact that our prototype uses poor algorithms in a few places, the testing runs were not particularly resource intensive. Ghostscript could be easily tested on any of the supplied inputs on a 32MB Pentium 100 laptop in about a minute or less.

- The preceding is in contrast to some earlier attempts which retained backwards-reachable objects. This appears to be ill-advised, since it is common to build data structures that reference a small number of "popular" objects (cf. [9]), which are referenced by many other objects, some of which are regularly dropped. Retaining objects "backwards-reachable" from popular objects can result in substantial space leaks for GC-robust applications. And there seems to be no obvious automatic mechanism for handling them specially.

- Our queue example exhibited spectacular growth in maximum object height, as expected. We stopped it once it passed a million. Interestingly, even this test was very easily runnable on the laptop, since the growing chain was not retained. In our test, the computed height grew without bounds, but the heap did not.

[7]We used the version from the Zorn benchmark suite. These are available from **ftp://ftp.cs.colorado.edu/pub/misc/malloc-benchmarks**. Our version was slightly modified in ways irrelevant to this discussion.

5. PRIOR WORK

We are not aware of any prior work attempting to give theoretical bounds on space usage with conservative collection.

There has been prior complimentary work examining more general issues related to space bounds for programs running with type-accurate garbage collectors (cf. [6, 1, 5]).

The observation that conservatively garbage collecting lazy functional programs may result in unacceptable space leaks is due to Wentworth [16]. We explored some of these issues a further, and provided techniques for reducing pointer misidentifications in [4]. The fact that Wentworth's observation also applies to a lesser extent to lazy functional languages collected with a precise generational collector is explored in [13]. Hirzel and Diwan recently completed a empirical study of the space space cost of conservative collection, by using a very interesting methodology [8].

6. SUMMARY

We introduced a notion of GC-robustness. We argued that most existing applications and all eager functional programs are already GC-robust. Any application in an imperative language can be made at least weakly GC-robust by clearing objects that would otherwise become chains of backwards-reachable objects. Typically this requires very little effort, since there are no or few such objects.[8] If it is necessary, such modification negates a small part of the convenience advantage of garbage collection. It does nothing to negate the safety advantages.

It appears that application data structures are quite likely to be naturally GC-robust, in contrast to, for example, the prohibition on cycles imposed by most reference-counting garbage collectors.

We suggested a testing mechanism to identify non-GC-robust data structures in programs, and aid the user in making an application GC-robust.

For a GC-robust application, we can bound the space that can be lost due to a single pointer misidentification or due to a single inappropriate promotion in a generational collector.

The worst-case space bound for a collector that scans even the full mutator stack conservatively is unpleasantly large, since every integer variable on the stack can introduce a false reference, and every false reference can retain a data structure the size of the accessible heap. Nonetheless, our results give some insight as to why substantial heap growth is rarely observed in practice.

For a slightly conservative collector such as that in [2], the bound on the space lost due to pointer misidentification appear to be on the same order as the factor of 10 to 20 that can be lost to fragmentation with a non-moving allocator in the worst case.[9] Since we routinely live with the fragmentation bound, perhaps there is hope for a practical application of our bound on retention due to pointer misidentification.

[8]Our experiment strongly suggests that Ghostscript usually generates none, even though that was clearly not a design goal. The fact that unbounded growth was not observed in [8] also suggests, though less strongly, that they are also rare in other applications.

[9]Any non-moving allocator may require a heap size that is $O(logk)$ times the size of live objects, where k is the size ratio between the smallest and largest object [11, 12]. Just as in our case, the typical case is ıfar better than that.

7. LIMITATIONS

For a collector that conservatively scans heap objects, there are other potential dangers that we have not addressed here. Even if all data structures are GC-robust, a bounded number of objects may be erroneously retained. This collection of objects may include a possibly greater number of additional misidentified pointers, which may cause additional retention, etc. Thus the total amount of erroneously retained memory may become unbounded even if all data structures are GC-robust. Even though properly reachable memory is bounded, the number of misidentified pointers is not.

This is arguably a less serious concern than non-GC-robust data structures, since the problem is about equally rare, depends more on statistical properties than individual pointer misidentifications,[10] is thus more amenable to conventional testing, and can usually be easily remedied by supplying a small amount of type information to the collector. It is usually far easier to communicate the layout of heap objects to a garbage collector than it is to communicate the location of pointers in registers, etc. Nonetheless, we know of no way to precisely bound the space usage of such programs without resorting to (usually somewhat dubious) statistical arguments.

Our space bound limits the memory treated as reachable by a conservative collector in terms of the memory reachable from proper pointers stored at run-time. We have not addressed the issue of defining what proper pointer values are visible to the collector at what point during program execution. In order to completely prove that a particular space bound is satisfied for a given source program, the language specification would need to define which objects are properly reachable at which point in the program. Most languages, *e.g.* Java, avoid doing so in order to avoid constraining the compiler as in [1]. Note for example that common subexpression on variable-sized objects can keep data structures reachable for extended periods, potentially increasing space usage without bounds.

Problems introduced by space-unsafe compiler optimizations tend to recur, in that if they apply to one invocation of a recursive function, they will apply to all. Hence our arguments about memory retention from a single spuriously retained object do not apply, and neither would any statistical arguments.

8. REFERENCES

[1] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] K. Barabash, , N. Buchbinder, T. Domani, E. K. Kolodner, Y. Ossia, S. S. Pinter, J. Shepherd, R. Sivan, and V. Umansky. Mostly accurate stack scanning. In *Proceedings of the Usenix Java Virtual Machine Research and Technology Symposium*, April 2001.

[10]In contrast to non-GC-robust data structures, this problem is likely to be reduced to infinitesimal occurrence probability by reducing the probability of pointer misidentifications, since unbounded growth requires an unbounded sequence of pointer misidentifications, not just a single unlucky integer value. Hence the black-listing technique of [4] is probably much more effective at addressing this problem than non-GC-robust data structures.

[3] J. F. Bartlett. Compacting garbage collection with ambiguous roots. *Lisp Pointers*, pages 3–12, April-June 1988.

[4] H.-J. Boehm. Space efficient conservative garbage collection. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 197–206, June 1993.

[5] D. R. Chase. Safety considerations for storage allocation optimization. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 1–10, June 1988.

[6] W. D. Clinger. Proper tail recursion and space efficiency. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 174–185, June 1998.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[8] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *Proceedings of the International Symposium on Memory Management 2000*, pages 1–11, October 2000.

[9] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In *Proceedings of the 1992 International Workshop on Memory Management (LNCS 637)*, pages 92–109. Springer, 1992.

[10] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[11] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, 1971.

[12] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):491–499, 1974.

[13] N. Rojemo. Generational garbage collection without temporary space leaks. In *Proceedings of the 1995 International Workshop on Memory Management (LNCS 986)*, pages 145–162. Springer, 1995.

[14] M. Serrano and H.-J. Boehm. Understanding memory allocation of Scheme programs. In *Proceedings of the 2000 International Conference on Functional Programming (ICFP)*, pages 245–256, 2000.

[15] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[16] E. Wentworth. Pitfalls of conservative garbage collection. *Software Practice and Experience*, 20(7):719–727, 1990.