singhAvineet

Summary on paper “Garbage Collection in an Uncooperative Environment”

1. Overview

Now a days most of the conventional compilers like javac are equipped with garbage collectors. The technique here describes storage collection and garbage collection in the absence of significant
cooperation from the code using the allocator. The approach is intended to simplify implementation of languages supporting GC.

In a generic automatic storage management, creation of an object / variable allocates associated space in the memory i.e. for an integer variable a size of 4 bytes is allocated in the memory. The least requirement here is to have enough information to distinguish references from other data which is normally addressed using tags. Like distinguishing the space allocated by pointer and a normal variable. There can also be a scenario to maintain the value of shared pointers which depends of reference count.

2. Approach

2.1 Overview

The approach is intended to avoid such cooperation due to following reasons-

- The primary goal was to implement programming language Russell on conventional hardware. This language stores variables in heap.
- Tagging of integers and floating points in memory is difficult.
- Support of GC with conventional implementations of common programming languages like C, Pascal, or Modula-2.
- It is difficult to design a compiler such that the generated code is guaranteed to preserve garbage collection invariants.
- Moreover, GC related bugs are intended to be the last to be removed from a new programming language implementation.

2.2 Description

The approach is intended to virtually have no cooperation from compiler generated object code like a conventional C compiler. The initial testing of the approach by removing certain object constraints and making them accessible was successful. The approach relies on the use of mark-sweep collector. This is based on the following important steps -

- o An initial pass traverses and marks all data accessible by the program.
- o A second pass returns inaccessible objects to an appropriate list of free memory blocks.
- o In order to determine accessibility, the mark phase of a mark-sweep collector must be able to locate all pointers in the user’s data.
- o Instead of requiring tags, any directly accessible data item is treated as potential pointer.
- o Storage allocator assures the object validity administered by the allocator. The object should be completely accessible.

2.3. Problems to be addressed

There are two implementation problems that need to be addressed-

- It is theoretically possible that an integer value may happen to correspond to the address of a valid object. It can however result in unnecessary memory consumption.
- The data structure should be efficient enough to recognize a valid object quickly.

3. Conservative Garbage Collectors

Conventional GC aims to prevent any potential memory leakage. This is achieved using reference counts. But this requires user program cooperation for each pointer assignment. The more common alternative is to have garbage collector determine accessibility of objects. This requires maintaining enough information to locate references contained in the registers. This is done using following approaches-

- Data item should contain enough information to identify it as pointer or non-pointer. This comes with significant cost of time.
- Reservation of certain memory spaces for certain classes of data. This may result to insignificant use of memory and registers.
- Use of traversal routine. Although this approach complicates the design of compiler and increases the probability of undetected errors.

The traversal routine is a partial solution. The type of a function is insignificant in determining the position of pointers. Thus, making it impossible to mark objects as reachable through function-valued variables without run-time tags. The polymorphic function F:

λx . cons(x, nil) can be assigned as $\forall t$. t $\rightarrow$ list(t)

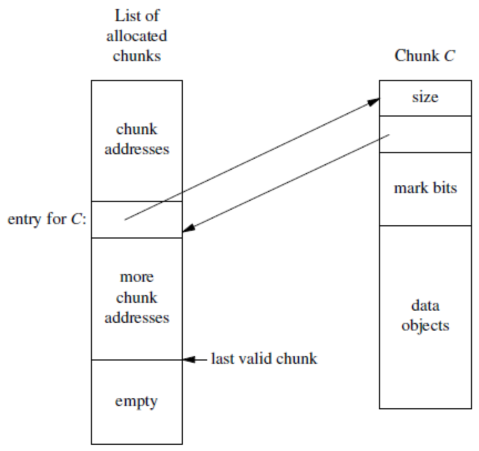
Although the solution poses above potential threats, yet it ensures that all storage that possibly cannot be assessed is reclaimed.

3.1 Algorithm

The algorithm ensures that anything other than a valid object is never marked as reachable. The approach avoids array initialization by maintaining a stack of pointers to initialized elements.

- A standard allocation scheme is used to obtain large chunks of memory. The chunks are always a multiple of 4k in size starting with 4 KB boundaries.
- Additional chunks are automatically obtained from the operating system if none are available.
- The allocator does not assume that memory obtained from the operating system is contiguous.
- It can coexist with other allocators.
- Smaller objects are allocated by maintaining separate free lists for each object size.
    - Allocation of small object consists of simply removing the first element of the free list.
    - Whenever an empty free list is encountered, a 4k chunk is obtained from the lower level allocator and subdivided uniformly into pieces of appropriate sizes.

    - Small object allocation is slower than simply advancing the pointer as it is done with compacting collectors.
- It is insured that objects in same region of memory have uniform structure with respect to nested pointers. Additionally, it is insured that objects in a given chunk have identical size.
- Sweep phase of allocator notices if a chunk is completely empty.
- If the chunk detected is empty, then entire chunk is returned to the chunk allocator.



- The chunk contains following information-
    - Size of objects in chunk.
    - Pointer to chunk is basically an array of all allocated chunks.
    - A space is allocated for mark bits in the chunk of objects.
- This chunk of information determines the address validity of the pointer as -\
    - If the address is below the lowest heap address or above the highest heap address, it does not correspond to an object.
    - If address corresponds to an object, then address of corresponding chunk of memory can be obtained.
    - Check if the offset of supposed object is a multiple of object size given by chunk header.
- Algorithms can be altered such that pointers to the interior of an object is recognized.

A minor disadvantage is that the resulting marking algorithm is unlikely to interact well with a data cache in the underlying machine. The only frequently accessed data is likely to be in the chunk headers. Several techniques are being used to minimize the probability of accidental generation of an integer which happens to be a valid heap address.

The experiment conducted on few workstations on code mostly written in C is positive.

4. The Russell Collector

- This a stack based non-recursive algorithm.
- The algorithm is structured to minimize stack growth for long, but shallow, linked lists.
- This suffices to mark objects accessible through registers, or through pointers aligned on 32-bit boundaries in the stack in context of Russell programs.

- As per an experiment conducted, the result was mostly positive.
- It is worth noting that the garbage collection time is more sensitive to the amount of *accessible* memory,
  than the amount of *available* memory. Garbage collection time is inversely proportional to the Available memory.

5. Major Contributions

This Garbage Collection technique has been implemented for a number of applications. The application in which this has been incorporated is mostly based on C. Its major contribution is with the Sun UNIX malloc utility. The utility is used primarily by the I/O library. This avoids the necessity of marking some permanent or explicitly freed data structures maintained, most significantly large I/O buffers.

This has been run in an interpreter for Fortran intermediate code supporting constructive real arithmetic. The interpreter generated a large amount of heap of size 2 MB. According to the result in sweep phase, the GC was measured to take about 0.4s/MB in heap. The mark phase took about 1.9s/MB of accessible memory in heap. This rounded up to the total time taken to around 1-3 secs.

The above approach was also used to add GC to large existing systems, TimberWolf and SDI.

- TimberWolf is a VLSI layout program consisting of 18371 lines of C code using dynamic storage allocation via malloc, calloc, realloc etc.
- SDI is a real-time video game, consisting of 5896 lines of C code, which uses dynamic storage allocation to manage objects moving around on the screen.

Neither of the two had to be re-compiled or changed only the allocator had to be relinked with the application. Although TimberWolf application ran without problem, SDI had 2 specific challenges which were subsequently fixed.

6. Drawbacks

Following are the major drawbacks of the above algorithm –

- Data item should contain enough information to identify it as pointer or non-pointer. This comes with significant cost of time.
- Reservation of certain memory spaces for certain classes of data. This may result to insignificant use of memory and registers.
- Usage of traversal routine complicates the design of compiler and increases the probability of undetected errors.
- The object should always be accessible for its memory to be free. This might not be the scenario always. To make it compatible for such programs, code modification is required.
- If the object pointed to is misreferenced then it may to its presence in stack for a long term although the scenario of its occurrence is very unlikely.
- Altering the algorithm so that the pointers of the object are recognized is an expensive check.

- The process of checking pointer validity is slower than simply checking a tag bit inside the supposed pointer.
- The marking algorithm is unlikely to interact well with a data cache in the underlying machine.

7. Conclusion

A collector similar to the above mentioned can be produced by one of the standard UNIX compilers. Provided the compiler should not provide additional accessibility issues. On using the above GC algorithm with two large existing systems TimberWolf and SDI both of which provides a large amount of Windows memory management using calloc, malloc, valloc, realloc. Neither of the two had to be recompiled or even the code needed to be changed. The two of them gave positive outcome with two hiccups which were gradually resolved. Any trace or evidence of memory loss was never witnessed as a result of the above algorithm.

The Debugging tool optionally invoked during collector run-time captures all the dynamically allocated memory information during run-time. The trace is very useful in debugging allocation problems arising with the garbage collector but is perhaps even more useful when the code being debugged was written without a collector in mind, and so should be free of memory ‘‘leaks’’.