Comp 411
Principles of Programming Languages
Lecture 28
Garbage Collection I

Corky Cartwright

April 8, 2022

Storage Retention Based on Tracing

True garbage collection relies on tracing all objects that are accessible from a set of designated "root" pointers. The roots are typically all pointers in the stack and static data area (where "global" variables are stored). In addition, when pointers are stored in registers, register pointers must be included in the root set.

Garbage collection is conservative; it must assume – in the absence of some form of proof – that every object accessible via a chain of references from the root set is still alive and must be retained after collection. In principle, compilers can perform static analysis (like type inference) to determine when some reachable objects (*e.g.* local variables) are never subsequently accessed. This information can be embedded in the object code and exploited by the garbage collector. To my knowledge, this technique has not yet been used in a production language platform.

Despite the fact that all garbage collection is conservative, the term *conservative garbage collection* is used for a special form of garbage collection where little if anything is known about which words in the stack are pointers. (Nearly all processors assume that addresses are aligned on the appropriate byte boundary. Our subsequent discussion relies on this assumption.)

Conservative Garbage Collection

A better name for this form of storage recovery might be *garbage collection in a hostile environment*. The canonical example of such storage recovery is revising the C/C++ `malloc` library to perform garbage collection. The `free` (`delete` in C++) procedure in such a library is a *no-op* (does nothing).

How does conservative garbage collection work? Brute force garbage collectors use a two stage process called *mark-and-sweep*. During the mark phase, the collector marks every reachable object by setting a designated bit in the header (which is reserved for this purpose during collection and is typically maintained in a "clear" state). After all reachable storage has been marked, the collector sweeps the entire heap (a contiguous area of memory) examining every object and linking unreachable objects into a free storage list (or other form of data structure recording freed storage blocks). Exact (non-conservative) collectors may also move objects to compact the footprint of live storage, but no object can be moved unless all pointers to that object can be identified and updated. Conservative garbage collectors cannot move objects in the absence of special constraints on what can point to the object to be moved. Conservative garbage collectors don't generally know whether an address aligned field in a heap object or stack frame is a pointer or not. So they assume that every such field is a pointer (to any live object) unless they can prove otherwise.

Reducing False Positives

The key to effective conservative garbage collection is minimizing the set of false positives, the assumption that a field is a pointer when it is not.

How can a conservative collector prove that a putative pointer is not in fact a pointer? By allocating data according to the following pattern: subdivide the heap into blocks containing objects of the same size. (Very large objects must be treated separately.) Given a putative pointer, the collector checks to see if it is properly aligned for the block that it points to. If objects are reasonably large on average, this technique is quite effective. Note that even when objects are not grouped by size but are address aligned, the collector can disqualify 75% (87.5% on a 64 bit machine) of non-pointers (on average) because they are misaligned (assuming a uniform distribution on non-pointer values in the heap).

In practice, conservative garbage collection has two serious performance disadvantages. First, it cannot compact the footprint of live memory because it cannot move objects. Second, if an application includes large circularly linked structures or arrays (assuming pointers can refer to elements), the chances of false retention of such objects after they are unreachable is high because a false pointer (*e.g*. an integer) pointing to any node of the structure forces its retention.

In addition, conservative GC requires that compilers not perform any optimizations that eliminate pointers to the bases of objects. C/C++ compilers generally comply with this restriction but it is not universally followed.

Conservative GC in Practice

Conservative GC is important technology. Xerox PARC ported their D-machine code bases (Dolphin, Dandelion, Dorado) to the Sun SPARC architecture in the late 80's using this technology. Many C/C++ applications like early editions of DrScheme/Racket rely on this technology.

An instructive variation on conservative GC is the design of early Sun JVMs. Since local variables in a Java stack frame do not have fixed types (a huge design error IMO), exact GC in Java is unnecessarily complex. The compiler or class loader must build "stack maps" for a sufficient set of "safe points" within the code for every method! (A safe point is designated point in the program where stack frame usage is known. Every point in the program is required to be a bounded distance – counting instructions in the execution stream – from a safe point.) As a result, early JVMs did *not* perform exact collection. They relied on the following trick: every object is accessed through a *handle* (a level of indirection) stored in a hash table exclusively under the control of the JVM. This level of indirection enables a conservative collector to safely move objects because *all* heap pointers are located in the hash table of handles. But the *cost* is enormous. The use of handles can slow the execution of compiled code by far more than a factor of two. Why? Accessing an object takes two memory references rather than one but the extra reference can cost far more than a single reference because of memory caching and pipelining. This design was a "quick and dirty" implementation hack.

Mark and Sweep Collectors I

The first production quality collectors (I am ignoring collectors for specialty languages like SNOBOL and APL that were generally interpreted) appeared in Lisp systems, which supported both interpreted and compiled code.

GC in Lisp systems was a much simpler problem that GC in Java, C#, or modern Lisp dialects like Scheme and Common Lisp. Why? The only dynamically allocated objects were *cons* cells (other than strings, symbols, arrays, and dynamically loaded machine code which were all treated specially and not stored in the main heap). In most machines, every *cons* cell occupied the smallest addressable unit containing two addresses. (In some old machine designs with 36-bit words and addresses no larger than 18 bits, a *cons* cell fit in a single word.) The key issue is that the heap was formatted simply as an array of *cons* cells.

Every *cons* cell included a designated bit (often the low-order bit of one of the word-aligned addresses in the cell) used for marking accessible cells. This "mark bit" was left clear (0) except during garbage collection. The "mark" phase of mark-sweep collection would simply performs depth-first search to mark all cons cells accessible from the roots. The "sweep" phase simply scans the entire heap (an array of cons cells) linking together unmarked cells in a free-list and clearing all mark bits. When memory was limited, the depth first search stack was embedded in the list structure being traversed by reversing pointers (this marking algorithm using this trick was called the "Schorr-Waite" algorithm).

Mark and Sweep Collectors II

In modern languages, mark and sweep collection is similar but more complex because heap objects vary in size. The “mark” phase is essentially unchanged (except that the traversal of an object's embedded pointers depends on the type of the object). But the “sweep” phase is more complex, because such collectors coalesce adjacent free objects. In addition, there can be holes (small unallocated fragments) in the heap, which can be detected by zeroing all unallocated memory and ensuring that the header word of an allocated object cannot be zero.

In practice, vanilla mark-and-sweep is not an effective approach to separating live data from garbage in modern languages because it is important to compact the live objects into a contiguous area of memory (for dramatically better cache behavior and simplified dynamic allocation). Hence, mark-and-sweep collection has been superseded by complex “mark-and-compact” schemes that compact the live objects while scanning. See Paul Wilson's monograph for the ugly details.

All mark-and-sweep collectors (including “mark-and-compact”) have a serious flaw that prevents them from being a good comprehensive solution to storage management in modern languages: *they run in time proportional to the size of the collected heap* (live data objects in the heap) – regardless of how few objects survive the collection.