Hardware and Operating System Support for Conservative Garbage Collection

Hans-Juergen Boehm
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Abstract

Conservative garbage collectors can automatically reclaim unreferenced storage, even if the client program is unaware of the existence of a collector. Nearly all garbage collectors can benefit from some form of hardware, and especially operating system support. Conservative collectors benefit particularly from two kinds of facilities. The first is trivial, but often overlooked; it should be possible to write a function that quickly identifies all data areas associated with a program.

Secondly, there should be an efficient mechanism for identifying areas of memory that have been recently written. This requirement is shared with other kinds of collectors. But conservative collectors, since they do not rely on moving objects can benefit more than moving collectors.

1 Introduction

A number of techniques can be used to reclaim memory once it is no longer needed. For short lived processes, it often suffices to reclaim the entire address space once the process terminates. In programming languages that provide no guarantees against arbitrary "memory smashes" it is usually considered acceptable to require programmer specified deallocation. It is possible to automatically count the number of references to memory objects, deallocating objects when reference counts reach zero.

There are substantial difficulties with all of these. Waiting for process termination is unacceptable for long lived applications, and especially for server processes. Explicit deallocation leads to subtle errors that are typically very difficult to diagnose. Reference counting collectors fail to reclaim cycles, and are error prone unless the compiler adds code to maintain reference counts. This may be infeasible or expensive.

Here we look at tracing garbage collectors. These wait until a suitable point in the computation (e.g. until memory is exhausted), and then identify, or trace through, all objects that are accessible by starting with program variables and following pointers. Any memory objects not traced in this process are guaranteed not to be subsequently referenced, and may thus be recycled by the allocator.

Tracing collectors can be further classified by whether or not they move memory objects, usually to compact all accessible memory objects into one or a few region of memory. Modern compacting collectors are nearly always "copying collectors", in that they copy accessible memory objects into a different memory region, rather than attempting to compact in place.

A tracing collector must be able to locate all pointer containing variables (roots). Furthermore it must be possible to follow pointers from one memory object to another. If objects are to be moved, we must guarantee that every pointer to every object can be identified, so that pointer values can be updated to reflect the new location of an object. If objects remain stationary, it suffices to find one path to each accessible object; other references to the object will not be disturbed so long as we don't erroneously collect the object. If objects are to be moved a misidentification of a nonpointer as a pointer is disastrous. Arbitrary non-pointer data may be changed. If objects are stationary, such a misidentification is relatively benign, and may at most result in failure to reclaim some memory.

This means that fully copying collectors, or collectors that may move any object, are restricted to environments in which close cooperation between the client (mutator) and the allocator/collector is possible. This cooperation may take the form of additional compiler generated tables describing data structure and stack layouts ([2, 11]), it may take advantage of tagged data, as in most Lisp systems, or...
it may require some other specific runtime data structures maintained with a combination of programmer and compiler effort (cf. [10]). In practice, such collectors are usually restricted to single-programming-language systems.

This paper concentrates on conservative collectors, or garbage collectors that require minimal cooperation from the mutator. Such collectors do not require that pointer identifications be exact. Typically any bit pattern that is a valid object address is taken to be a pointer. Thus objects may in general not be moved. We will assume that no objects are moved, but this restriction is sometimes stronger than necessary. (A collector that moves some objects is described in [4].)

As is discussed in [5], this style of garbage collector may be used with most unmodified C programs. The Xerox Portable Common Runtime [17] (PCR) relies on this approach to provide a common garbage collection facility for programs written in C, Cedar, Modula-3, Scheme, and other programming languages.

In spite of the fact that we explicitly try to avoid compiler dependencies in order to facilitate interoperability, such collectors generally benefit substantially from certain kinds of rather specific operating system support. The collector of [5] illustrates what can be done with only a very ordinary collection of operating system facilities. Its deficiencies are primarily in the following two areas:

1. Portability. The primary issue here is that of identifying a sufficient set of memory regions, such that all reachable objects can be found by tracing from these regions. Typically these will include the stack, register, and static data areas. Unfortunately, there is usually no portable way for a garbage collector to find these. Furthermore, in a multi-threaded environment, certain combinations of hardware and compiler features make it difficult to find a complete set of roots at certain program execution points.

2. Performance in large address spaces. The simple collector of [5] must stop all mutator execution while it traces through all reachable heap storage. In large address spaces, this can result in the mutator pausing for several seconds or, in the presence of paging, even for several minutes. As described in [6], this can be avoided using a combination of two techniques. First, we use a generational collection algorithm, that usually traces only through recently allocated storage. Second, we use a mostly parallel collector, that performs most of its work concurrently with the mutator. Both of these rely on support from the virtual memory system. Unfortunately, this support is again either lacking or very marginal in most current commercial systems.

Interestingly, and unlike for collectors relying on tagged pointers, it does not appear that the addition of specific processor instructions would have helped matters significantly. A compare-and-swap instruction could probably be used to reduce allocation overhead in a multi-threaded system. An integer divide instruction allows fast simple pointer validity tests. Bit vector operations would speed up mark bit operations. But none of these is likely to have much more than a 10% impact on object allocation/deallocation round-trip time. The last two can largely be sidestepped by cleverer programming. Thus the emphasis of this paper is on operating system and virtual memory facilities.

The following section addresses the area of portability. Next we discuss virtual memory support required for parallel and generational collection. Section 4 discusses the somewhat unusual scheduling requirements of parallel garbage collectors.

It is worth noting that, at least with respect to virtual memory support, we require a proper subset of what is required by other common algorithms [3]. In particular, we do not require the ability to handle page faults at user level, provided we can determine whether pages are dirty. Thus the ability to quickly handle traps in user mode is secondary for us, unless we are forced to use it to simulate other, simpler facilities.

2 Portable Root Finding

One of the most difficult practical problems in the implementation of any tracing garbage collector is the identification of roots, i.e. program variables containing pointers. In the case of a conservative collector, it suffices to identify a collection of memory locations that contain a superset of all roots. However, we would like to identify roots without requiring specific support from the language compiler, or sometimes even from its runtime system. We constructed and distributed a library that defines a garbage-collecting replacement for malloc [5]. It should be possible to link a C program against this library with no other changes, and have the garbage collector in the library locate the
roots for the mutator C program. We encountered two difficulties in accomplishing this.

First, there are no standard primitives that allow the garbage collector to locate the regions in memory that contain statically allocated program variables, the program stack(s), or that allow the collector to copy processor registers to memory. (In our experience, most operating systems do not provide any primitives for doing this, standard or not. The C setjmp primitive can be used to copy the registers on some machine, but not on all machines.) This problem is becoming more substantial as process address spaces become less contiguous. Shared libraries often result in fragmented address spaces, with statically allocated variables distributed over several regions of a process’ address space.

The second problem is probably more significant, but is much more related to optimizing compilers than to operating system issues. Consider a machine such as IBM’s RS/6000 which allows addressing with a signed 16 bit displacement, and provides an instruction to add a 16 bit quantity to the upper half of a 32 bit word. Assume that the variable x points to a structure that has field a at a displacement of 40000. With optimization enabled, the C statement

\[ x = x + a; \]

will be compiled as (in pseudo-C where + denotes machine addition):

\[ x += 65536; \quad x = *(x - 25536); \]

If x is the last reference to an object of size less than 65536, and a garbage collection is triggered between the above two instructions, a collector would have no opportunity to find a pointer to either the new or the old value of x. Similar problems arise of preincrement addressing is used to traverse an array in the heap. (Debuggers have similar problems if the dereference fails. Hence these problems typically do not arise with unoptimized code.)

It is hard to see how this problem could be solved by any means other than requiring minimal cooperation from the compiler optimizer.

But it does have implications for hardware and operating system design. The compiler should be able to generate safe code for the above statement that is not appreciably worse than the code given above. (On most modern machines, including the RS/6000, this is already true. The identical code sequence in which a different register is used for the intermediate result is safe. This might not be true if the machine provided only a two operand add instruction.)

In a multi-threaded system, it may be advantageous to provide a convenient and portable facility for single instruction execution. With such a facility, the collector could advance each thread of execution to the end of the current basic block before beginning a collection. This would simplify the compiler’s job in that we would no longer require that pointers be recognizable in the interior of a basic block. Thus the above example would become safe, though it would still be unsafe if an optimizing compiler decided to separate the two instructions into two distinct basic blocks. The weaker safety requirement should result in better performance on register-poor machines. Since usually the only effect of the safety requirements is to increase the number of live registers, it could be expected to have a minimal effect on machines with large register sets.

Note that a portable facility for single-step execution is even more likely to be required by a garbage collector that requires precise pointer identification, and expects compiler cooperation. Such a collector is likely to require that mutator threads be executing at one of a small number of execution points for which a complete description of register contents is available.

Finally, it is important to keep in mind that conservative collectors are most effective if nonpointer data rarely has the same bit representation as a valid pointer. Large, sparsely occupied address spaces clearly help. (See [18] for a discussion of conservative collection in dense address spaces.)

Certain architectural features have a more subtle effect on false pointer identifications. Requiring pointers to be aligned on word boundaries helps the collector, since it greatly reduces the number of bit patterns that must be examined as potential pointers. Frequent use of small or misaligned integers hurts, since concatenations of these are much more likely to show similarity to pointers than single integers. It’s sometimes possible to partially compensate for these effects through clever heap placement, but that is not a complete solution.

Architectures encouraging large procedure call frames may introduce additional false references, especially if large parts of the frame are not properly initialized. Architectures with large register windows (e.g. SPARC) exhibit this problem. A “random” value in a “new” register window, is likely to eventually migrate to the stack and, depending on the source of the “random” value, may have a high probability of appearing to be a pointer.
3 Virtual Memory Support for Better Response

A simple conservative garbage collector, such as the one described in [5] operates in the following stages. We assume that each object has an associated mark bit, which is set when an object is found to be reachable.

1. Stop the mutator.
2. Clear the mark bits.
3. Start at the roots, and follow all "pointers", until all objects reachable by the client are traversed. Set the mark bits on all objects encountered in this process.
4. Traverse the heap, adding unmarked objects to whatever free list structure is used by the allocator.
5. Restart the mutator.

The problem with such an approach is usually not object allocation and deallocation time, unless the client code is much more allocation intensive than most C code. Even such a simple collector can sometimes outperform a malloc/free style allocator. (The most common reason for this is a mediocre malloc/free implementations. A more substantive reason is that it is cheaper to deallocate objects en masse than one-at-a-time.) The difficulties we encounter are normally pause times, perhaps aggravated by paging during garbage collections.

Steps (2), (3) and (4) together consume an appreciable amount of time, during which the mutator program is completely stopped. For small, single-threaded processes, whose only real-time constraint is reasonable interactive response, this may be acceptable. For a process with 1 MB of live data running on a SPARCstation 2, delays are on the order of .5 seconds, and are most likely to occur during compute-bound phases, when they are not noticeable. For larger systems, this is often unacceptable.

It is relatively easy to avoid stopping the client program during step (4) of the collection. We simply perform this step incrementally as part of the call to the allocator. It is also not difficult to arrange things such that a page is unlikely to get traversed until just before it will be allocated on. This keeps step (4) from touching any additional pages.

Step (2) could be handled similarly. However, if mark bits are allocated in a separate bit map, this step is typically fast enough that it is not an issue in any case.

Step (3), the mark phase, is a much more subtle issue. Two techniques are available to us. First, we can build a generational collector, that avoids tracing through all reachable objects during every collection.

3.1 Generational Collection

More recently allocated objects are much more likely to become unreachable quickly. Hence it makes sense to concentrate our effort on such objects. This is approach is common for copying collectors (cf. [16]). A general technique for building generational, conservative collectors is described in [7].

All such techniques assume that old objects, that is objects that were not recently allocated, and that are not candidates for collection, are live. Thus in addition to marking young objects reachable from roots, we must also mark young objects reachable from old objects. Typically the collection of old objects is much larger than the collection of young objects. Thus this technique can be very efficient, so long as we have a fast technique for finding references from old objects to young objects.

One fast way to find such references is to remember the list of such references at the last collection, and then to rescan objects that were potentially written to in order to find newly introduced references. Potentially written objects can be easily located if the operating system provides a primitive that identifies virtual memory pages that were written since the last query [14, 3].

The above stop-the-world collector can be particularly easily turned into a simple generational collector. For most collections, we delete step (2), leaving mark bits set for objects that survived the last collection. This means that only objects allocated since the last collection will be considered for reclamation. In addition to marking from roots, we must then start marking from previously marked objects that reside on dirty pages, i.e. on pages that were written since the last collection. This greatly reduces the cpu time consumed during such a collection (typically by a factor of 3 or so in the Xerox PARC PCedar environment). Perhaps even more importantly, we essentially only read pages that were written during the last collection cycle, thus avoiding potential paging problems.

3.2 Parallel Collection

A second technique for reducing pause time during the collection mark phase is to run it concurrently...
with the mutator. Traditional techniques for concurrent garbage collection (cf. [15, 9]) are not directly usable, since they require elaborate cooperation from the mutator.

We again need the help of the virtual memory system. In [6], we describe a general approach to parallelizing garbage collectors that is particularly appealing in our environment. The idea is simply to let the mark phase proceed with the mutator running, and no synchronization between the two. This is well-known to produce incorrect results. However, if we keep track of objects that were possibly written to by the mutator while marking was in progress, it is easy to correct such mistakes. In fact, it can be shown that the partial collection algorithm from above will do so, and will run significantly faster than if it were run by itself. Thus even a full collection, i.e. one that considers all objects for possible reclamation, can be run as follows:

1. Clear the mark bits.
2. Reset all pages to “clean”, i.e. not written to.
3. Stop the mutator.
4. Mark from roots.
5. Mark from marked objects on “dirty” pages and from roots that are not on “clean” pages.
6. Restart the mutator.
7. Traverse the heap, adding unmarked objects to free list structure.

Thus both generational and parallel collection requires access to information about which sections of memory have been written. Such information is usually available to the virtual memory manager in the operating system. Hardware provided dirty bits can be used to efficiently track such information. A small amount of additional work is needed to maintain the information for disk resident pages, and to maintain a shadow copy of the dirty bits, so that the user program can clear them. Unlike a well-known technique for parallel copying collection [1], we do not require the ability to intercept page protection faults at user level, though such a facility can be used to obtain dirty page information.

Unfortunately, few commercial operating systems provide the user program access to dirty page information. The PCR collector currently write protects pages and catches write fault signals in order to compute its own dirty page information. Unfortunately, the processor overhead associated with this approach is such that the signalling and protection overhead (on the order of 500 faults per collection at 500msecs per fault with collections at 1 MB intervals), accounts for about a third of the total cpu time spent on garbage collection. A second problem with this approach is that most UNIX implementations do not allow arbitrary system calls to be restarted after a protection fault. This means that PCR must be careful to ensure that memory is not protected before a system call is issued. This is difficult for system calls with an open ended argument format, such as the UNIX ioctl call.

A single system call that returns and logically resets dirty bits on virtual pages would be far preferable. Reasonable performance of this system call is important, since it must at times be executed with the mutator stopped. On a SPARCstation 2, we often see collection pauses on the order of 100 msecs. It must be possible to retrieve the dirty bits for a 20 MB address space in significantly less than that.

### 3.3 Other VM Support

In addition to access to dirty bit information, certain other kinds of virtual memory support are useful. If a parallel collector is to run in a separate heavy-weight process, then it clearly needs an efficient way to share the heap with the mutator.

Essentially all garbage collectors can benefit from a means of informing the virtual memory system that the contents of a certain page are no longer of interest, and could safely be replaced by a zero-filled page. This prevents garbage pages from being written to disk. This is particularly important for copying collectors, since large regions of memory may be known to contain garbage. However, even a nonmoving collector is likely to recover large chunks of garbage at once [12]. In general, objects allocated together are likely to become garbage together. Thus even if only very small objects are allocated, the garbage collector will often recover groups of entire pages. In normal Cedar use, we often see 10-20% of pages that were used for small object (at most 1/8 page) allocation in a given collection cycle completely empty by the time of the next collection. This fraction is occasionally as high as 40%.

### 4 Process Scheduling and the Collector

Parallel garbage collectors require some control over process scheduling. So long as there is more than one
thread of execution, essentially any practical collector will need a facility for stopping the mutator process(es), at least for short periods. At a minimum, it needs to obtain a consistent snapshot of the mutator stack(s). In the case of the collector described in [6], a little more work needs to be done with the mutator stopped. Nonetheless, the mutator is often stopped for less than 100 msecs. Thus it is important to be able to stop and restart the mutator in significantly less time than this.

If the collector is running in a UNIX-like heavyweight process, then sending signals requesting the mutator processes to stop is usually sufficient. If the collector is running as one of many lightweight threads, then a more direct facility for stopping all other threads may be necessary for adequate performance. The PCR threads package provides such a facility.

A parallel collector running concurrently with mutator allocation should ideally be guaranteed make progress at least in proportion to the amount of mutator allocation, without preventing mutator progress for extended periods. If this is not guaranteed, and the garbage collector fails behind, then allocating mutator threads may have to be blocked waiting for more memory to become available, or we may have to allocate an unreasonably large heap.

This implies that a strict priority-based scheduler is not sufficient for scheduling the collector. Ideally we would like the collector to obtain a fixed percentage of processor resources plus resources proportional to the number of intervening allocations. This should be independent of the priorities of other threads.

In PCR, this kind of scheduling is approximated by running a high priority thread that alternately sleeps for the rest of its time slice, or yields the rest of its time slice to the garbage collector. The fraction of time slices it gives to the collector depends on the amount of allocation since the beginning of the collection. This relies on the fact that the thread switcher provides a call to the garbage collector that allows a garbage collector thread to yield to another specific thread.

More interestingly, garbage collectors in general, and conservative garbage collectors in particular, require facilities that are otherwise in low demand, and are thus often overlooked. Most importantly, a high performance garbage collector needs to be able to narrow it's search for recently updated pointers. In some cases, this can be done explicitly by the mutator. In the case of a conservative collector requiring no mutator cooperation, this requires an efficient facility for recognizing recently written virtual pages, ideally with a moderate page size. Most current operating systems provide at best marginally usable facilities for this.

If the collector implementation is to be portable, then there must be a standard facility for locating nonheap memory that might contain mutator data, and must thus be treated as part of the root set by the collector. UNIX-like operating systems appear to be regressing in this area. Early implementations tended to allocate different segments contiguously, with loader defined symbols delimiting them. More modern implementations often load programs into discontiguous regions of the address space, and either provide no facility, or a highly nonstandard facility for finding these regions.

Garbage collectors require control over the scheduler, both to inhibit scheduling of mutator threads or processes, and to allow a parallel collector to obtain its proper share of the processor, independent of mutator priorities. The former can often be accomplished with an asynchronous signalling facility. The proposed IEEE POSIX thread standard [13] provides enough facilities to stop threads, provided it is acceptable to broadcast signals to each thread, and the mutator code is sufficiently well-behaved to guarantee that the signal will be handled in a reasonable amount of time. But the same proposal does not appear to provide a facility for ensuring that both mutator and collector make reasonable progress, independent of mutator priorities. In particular, it does not provide a facility for yielding control to a specific thread, which would essentially enable a high priority collector thread to perform the right kind of scheduling.

5 Conclusions

A reasonably sophisticated garbage collector is likely to be a demanding client of process switching, synchronization, and virtual memory facilities. In some cases, the allocator data structures are the most commonly accessed shared resource. Thus the allocator/collector is particularly sensitive to synchronization overhead.

Acknowledgements

Alan Demers suggested, and first built a PCR implementation of a user level scheduler based on directed yields to another thread. Parag Patel pointed out that setjmp can often be used to obtain copies of machine register contents for use in marking.

UNIX is a Trademark of AT&T Bell Laboratories. RS/6000 is a trademark of International Business Ma-
chines Corporation. SPARC and SPARCstation are trademarks of either Sun Microsystems or SPARC International.

References


