Garbage Collection in an Uncooperative Environment

Hans-Juergen Boehm

Computer Science Department, Rice University, Houston, TX 77251-1892, U.S.A.

Mark Weiser

Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, U.S.A.

A later version of this paper appeared in Software Practice and Experience 18, 9, pp. 807-820.

Summary

We describe a technique for storage allocation and garbage collection in the absence of significant cooperation from the code using the allocator. This limits garbage collection overhead to the time actually required for garbage collection. In particular, application programs that rarely or never make use of the collector no longer encounter a substantial performance penalty. This approach greatly simplifies the implementation of languages supporting garbage collection. It further allows conventional compilers to be used with a garbage collector, either as the primary means of storage reclamation, or as a debugging tool.

Our approach has two potential disadvantages. First, some garbage may fail to be reclaimed. Second, we use a "stop and collect" approach, thus making the strategy unsuitable for applications with severe real-time constraints. We argue that the first problem is, to some extent, inherent in any garbage collection system. Furthermore, based on our experience, it is usually not significant in practice. In spite of the second problem, we have had favorable experiences with interactive applications, including some that use a heap of several megabytes.

Keywords Garbage collection Storage management Debugging

Introduction

Conventional automatic storage management systems rely on the user program or, more precisely, the object code generated from the user program, to maintain certain invariants necessary for the correct operation of the garbage collector. At a minimum, enough information is maintained to allow the collector to distinguish references from other data. Normally this is accomplished by maintaining tags. For example, pointers might be constrained to have a fixed bit pattern in the low order bit positions. Non-pointers, such as integers, would be required to have a different bit pattern in the same position. Register contents might obey a slightly different convention.

In more ambitious systems the user program may be required to maintain reference counts, or even to update marks used to determine the accessibility of data structures[15].

We wanted to avoid the need for such cooperation, for the following reasons:

1. Our primary goal was to implement the programming language Russell[16,5] on conventional hardware. The language allows arbitrary functions to be treated as data. In the general case, this forces activation records and/or closures, as well as variable cells to be heap allocated. However, a number of relatively simple compiler optimizations are frequently successful in reducing this to a mini-

---

UNIX is a trademark of AT&T Bell Laboratories. Sun 3 and SunView are trademarks of Sun Microsystems. VAX is a trademark of Digital Equipment Corporation.

mum[25,10,6,11,20].

For programs written in a Pascal-like style, the total amount of time spent in garbage collections is thus small. Since garbage collections are still *possible*, it is nonetheless necessary to preserve any information needed for storage management. In order to obtain overall execution times comparable to Pascal code, it is essential that such bookkeeping also be reduced to a minimum. We would like to compile a Pascal-like subset to conventional machine code, and pay for the garbage collector only when it is actually needed.

2. In any practical programming language implementation, compatibility with the underlying operating system and existing program libraries is an issue. Communication with these routines normally requires that we be able to manipulate standard machine representations of data. For example, offsets into a file are normally represented as 32-bit integers. If our garbage collector requires that all integers have, say, the least significant bit equal to 0 to distinguish them from pointers, it becomes difficult to communicate file offsets to the operating system; typically interface routines must be provided to perform the appropriate translation. In general, any tagging scheme requires nonstandard data representations, and thus introduces similar problems.

   A related problem is that standard subroutine libraries cannot be relied on to preserve garbage collector invariants. This problem is particularly severe if such routines perform their own storage allocation, *e.g.* for I/O buffers.

3. Requiring integers to be tagged either reduces the effective size of integers, or requires extra memory to be allocated and manipulated as part of each integer[17]. Typically this slows down some operations on integers. Tagging of floating point values is normally even more difficult.

4. We would like to support garbage collection in conjunction with conventional implementations of common programming languages like C, Pascal, or Modula-2.

5. It is difficult to design a compiler such that the generated code is guaranteed to preserve garbage collection invariants. It is hard to isolate problems dealing with incorrect maintenance of, for example, tagging information. Such errors in the object code can potentially go unnoticed for long periods of time. Even when they do appear, the problem is difficult to isolate, since symptoms usually appear a long time after the erroneous code was executed. Garbage collection bugs tend to be among the last to be removed from a new programming language implementation.

Here we describe an approach to garbage collection that requires virtually no cooperation from compiler generated object code. It can be used with code produced by, say, a conventional C compiler (provided users do not go out of their way to conceal references to objects). This approach has been used to support a Russell implementation for Sun 3 workstations. A slight variation on the same garbage collector was supplied to students in support of a compiler class project requiring storage management in the runtime system. Again, this was made feasible by the lack of constraints on the object code produced by the (student-written) compilers. Finally, we used this approach to add automatic storage management to two very large C programs written by others. Both ran unmodified using our collector.

Our approach relies on the use of a mark-sweep collector. An initial pass traverses and marks all data accessible by the program. A second pass returns inaccessible objects to an appropriate list of free memory blocks. (See Reference [13] for a survey of garbage collection algorithms.) Unlike a compacting or copying collector[4], accessible data is never moved.

In order to determine accessibility, the mark phase of a mark-sweep collector must be able to locate all pointers (references) in the user's data. Instead of requiring tags, we treat any data item directly accessible to the program (*e.g.*, the contents of a register or a word on the stack) as a potential pointer. The storage allocator assures that given such a data value, we can determine whether it points to a valid object that is administered by the allocator. If so, we assume that the data value in fact was a pointer, and that the object it points to is accessible. Similarly, we treat any data inside the objects as potential pointers, to be followed if they, in turn, point to valid data objects. A similar approach, but restricted to procedure frames, was used in the Xerox Cedar programming environment[19].

There are two implementation problems that need to be addressed. First, it is theoretically possible that an integer value may happen to correspond to the address of a valid object. In contrast to a copying collector, such an error cannot result in incorrect program execution. It can however result in unnecessary memory consumption. Though we cannot completely avoid the problem, the probability of such a misidentification can easily be made very small. We discuss the issues and techniques involved in the next section.

A second implementation issue is the data structure used by the collector. Unlike a conventional collector, it is essential that we can recognize a valid object quickly. The third section describes the data structures used by our collector. We conclude with a summary of our experiences.

"Conservative" Garbage Collectors

Most conventional storage management systems attempt to insure that all garbage, i.e. inaccessible storage, will eventually be reclaimed, and returned to a pool of available storage. In the absence of circular structures this can be accomplished by maintaining reference counts[14]. This requires user program cooperation for every pointer assignment.

The more common alternative is to have a garbage collector determine accessibility of objects. This requires maintaining enough information to locate references contained in the registers, on the execution stack, or nested inside other data objects. Most frequently this is done using a combination of the following three approaches:

- Each data item may contain enough information to identify it as either a pointer or a non-pointer. This usually results either in a loss of address space and a loss in the usable size of machine integers, or in a larger size for each data item[17]. On general-purpose architectures, it usually also involves a significant cost in execution time.
- Certain areas of memory may be reserved for certain classes of data. For example, we may insist that pointers reside in a subset of the registers. Similarly, we may divide the heap into different regions, such that all the objects in each region have the same type and thus an identical storage layout.

Unfortunately this approach is not directly applicable to the execution stack, since the position of pointers in the stack changes with time. (See reference [12] for an interesting solution in a combined reference counting and garbage collection environment.) Other problems are that it may result in inefficient use of memory and registers, and, as described below, even in a statically typed language, it may not be possible to determine the complete layout of an object at compile time. Thus run-time decisions may be needed to place an object in a proper region.

- In the presence of static typing information in the source program, it is possible to generate a traversal routine specifically tailored to a user program[8]. Such a routine would depend on the particular data structures declared by the program, and would embody knowledge of pointer positions within those data structures. Unfortunately, such an approach further complicates the design of the compiler, and increases the probability of undetected errors.

  More importantly, this is only a partial solution. The type of a function is not enough information to determine the position of pointers in its activation record (or closure in some implementation schemes[10,25]). Thus it is not possible to mark objects reachable through function-valued variables without run-time tags. (At some cost in garbage collection time and compiler complexity, code addresses could double as tags.)

  Polymorphic functions (functions operating on data of more than one type) further complicate matters. In type systems similar to that of ML[18], the function F:

  $$\lambda \mathrm{x}\ .\ \mathrm{cons}(\mathrm{x}, \mathrm{nil})$$

  would be assigned type

  $$\forall t\ .\ t \rightarrow \mathrm{list}(t)$$

  (F is a function that, for any type t, maps an object of type t to a list, whose elements have type t.) If the execution of 'cons' should happen to invoke the garbage collector, there is no apparent way to determine the structure of x, and thus to traverse x. In fact, the structure of x may differ from one call to the next. Clearly some tag information is needed. (A relatively clean way to handle this information, especially in the context of the Russell type system, is to explicitly pass the type t of x as a second parameter, and include the traversal routine for x as part of the representation of t.)

In return for the overhead associated with these schemes, both in execution time and in demands on the compiler, we can insure that all objects not accessible from the run-time stack, machine registers, etc., will be reclaimed. However, this is clearly not the property desired by the programmer. We would like to insure that all storage that cannot possibly be accessed *by the program* is reclaimed. These notions may differ for several reasons:

- For reasons of execution efficiency, the compiled code may fail to destroy obsolete references, *e.g.*, it may fail to explicitly clear a register dedicated to pointer values after the last reference to the register. Even if such optimizations are not made intentionally, it is possible that they will occur accidentally, since such a "bug" in the compiler is virtually undetectable.

- A particular run-time representation of environments (identifier bindings) may involve unnecessary references not intended by the programmer. For example, a particular implementation may represent the current environment as a linked list of activation records, each containing values corresponding to identifiers declared in a given procedure or block[3]. Consider the program in Figure 1. When execution reaches the label *A*, the current environment could be represented as in Figure 2.

  It is evident from the program text that the values associated with *a* and *b* cannot be subsequently accessed. On the other hand, both are clearly reachable by following pointers from the global (and current) activation record. (Moving local variable declarations, such as those of *a* and *b* to the surrounding procedure activation record, makes matters still worse.)

A different run-time structure[10] could avoid the problem of Figure 2. But the fundamental difficulty is that whether an item *a* can subsequently be accessed as a result of some combination of user input is not decidable; any programming language implementation *must* use a criterion for accessibility that will occasionally err on the conservative side. In a sense, we are not qualitatively changing the correctness assertions one can make about the garbage collector; we are merely accepting the fact that any garbage collector may fail to reclaim memory that can never be accessed. As a result we are free to make reasonable trade-offs between the likelihood of excess storage retention and certain other kinds of garbage collection overhead.

Algorithmic Issues

In order to safely avoid tagging pointers, we must be able to determine whether an address corresponds to the beginning of an object maintained by the collector. This test has to be sufficient to insure that we *never* mark anything other than a valid object as reachable, since setting the ‘‘mark bit’’ corresponding to something else may destroy arbitrary data. Our approach is based on a well-known technique for avoiding array initialization by maintaining a stack of pointers to initialized elements[2].

```
let
   f =  let
           a = ...
           b = ...
        in
           if ... a ... b ... then (λy . y) else (λy . y + 1)
in
   A: ...
```

Figure 1: An opportunity for unnecessary retention

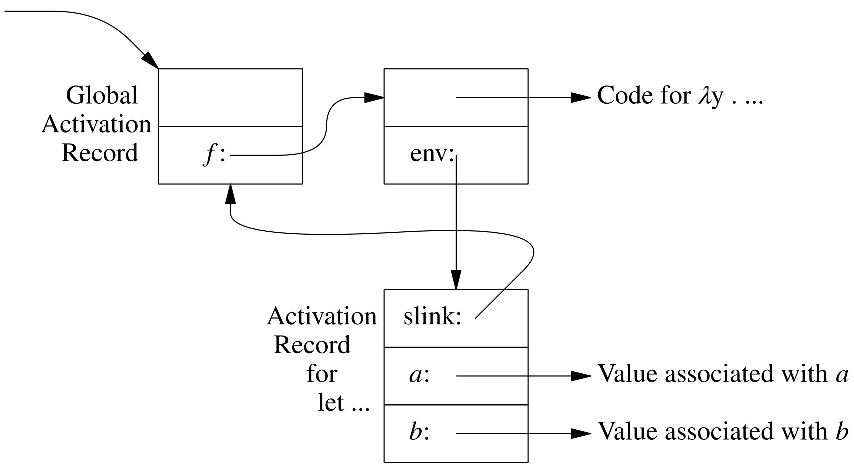

Figure 2: Environment representation for Figure 1

The allocator itself is structured as follows. A standard allocation scheme is used to obtain large "chunks" of memory. Such chunks are always a multiple of 4K in size, and always start on 4 Kbyte boundaries. Additional chunks are automatically obtained from the operating system if none are available, or if a garbage collection failed to reclaim a sufficient amount of free space.

The allocator does not assume that memory obtained from the operating system is contiguous. It can coexist with other allocators, so long as the garbage collector does not need to consider references *from* objects managed by a foreign allocator.

Smaller objects are allocated by maintaining separate free lists for each object size. In the normal case, an allocation of a small object thus consists simply of removing the first element of the appropriate free list. This typically involves the execution of four or five machine instructions, including the test for an empty 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 the appropriate size. We do not insist that objects in the same region of memory have uniform structure with respect to nested pointers, but we do insure that all objects in a given chunk have identical size.

Small object allocation is somewhat slower than simply advancing a pointer, as is normally done with compacting collectors[4]. However, it does make it *possible* to explicitly deallocate storage to avoid garbage collection overhead. The Russell run-time system makes occasional use of this to free objects known to be inaccessible. Similarly, the compiler can usefully insert explicit deallocation calls when static analysis determines that an object is no longer accessible.

The sweep phase of the allocator is written so as to notice if a chunk is completely empty. If so, the entire chunk is returned to the chunk allocator, instead of returning the individual objects to the appropriate free list. Adjacent free chunks are coalesced.

Located at the beginning of a chunk C is a header containing the following information:

- The size of objects in the chunk. For large objects, the size is that of the single object contained in the chunk.
- A pointer to the entry for C in a contiguous list of all allocated chunks. The entry in the list is simply a pointer back to C.
- An area reserved for mark bits corresponding to the objects in the chunk.

Figure 3 depicts the data structure associated with the chunk C.

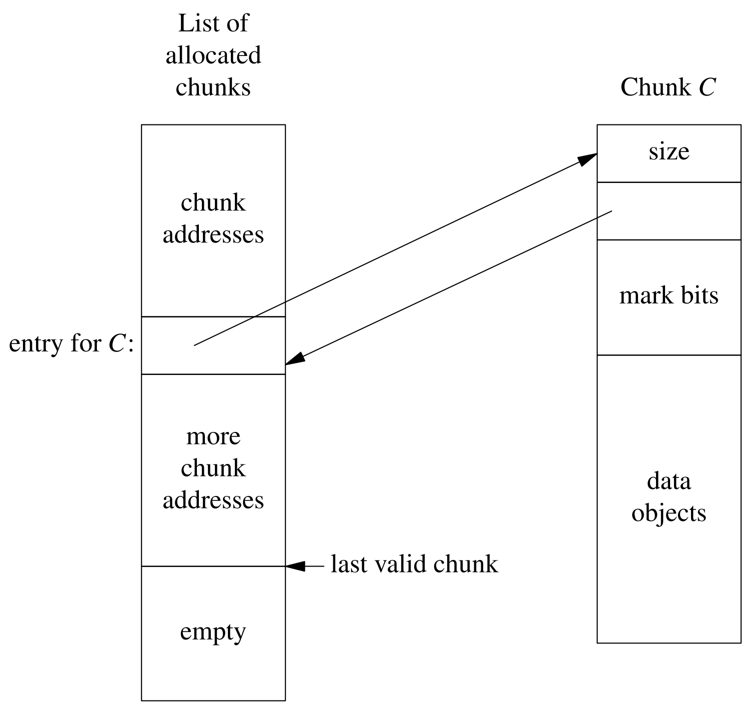

Figure 3: Recognizing valid chunks

It is now possible to determine whether a pointer corresponds to a valid address as follows:

1. If the address is below the lowest heap address or above the highest heap address, it does not correspond to an object.

2. If the address corresponds to an object, the address of the corresponding chunk can be easily obtained by masking out the 12 least significant bits of the address, corresponding to the 4 Kbyte alignment of chunks. We can check whether the address a so obtained actually corresponds to a chunk by first checking that the pointer to the list of allocated chunks contained in the supposed chunk header is within range, i.e. it is an address in the range currently occupied by the list. If so, we check that the entry in the list matches a. If it does, we know that the chunk address is valid, since only legitimate chunks appear in the list.

3. Finally, it is necessary to check that the offset of the supposed object from the chunk header is a multiple of the object size given by the chunk header, and that the object does not extend past the end of the chunk.

We assume that for every accessible object there is an accessible pointer to the beginning of the object. A pointer to the interior of an object is always viewed as invalid. In the case of small objects that fit entirely into a single chunk such pointers would fail the third test. In the case of a pointer to the interior of a large object it is also possible that the chunk address obtained in step 2 is invalid.

It is possible to alter the algorithm so that pointers to the interior of an object are recognized. This requires a relatively expensive check (*e.g.* a hash table look up) to see whether an apparent reference to an invalid chunk is actually a pointer to the last part of a large object. As is discussed in the next section, this is usually not a significant problem, since such references are normally rare.

The process of checking pointer validity is slower than simply checking a tag bit inside the supposed pointer, but not outrageously so. In particular, the running time of a single check is bounded by a small constant, independent of the size of memory.

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. Since the least significant bits of all chunk header addresses are the same, headers are likely to get mapped to a very small number of cache addresses. Thus, few of them will be present in the cache at any given time.

We use several techniques to minimize the probability of accidentally generating an integer that happens to be a valid heap address. Most importantly, the process address space is laid out in such a way that small integers never constitute valid heap addresses. Since we are operating in a UNIX environment, this is automatic.

We further distinguish between the allocation of *atomic* data that never contains embedded references (*e.g.*, very large integers and character strings), and *composite* data that may contain embedded references. Separate free lists and allocation routines are provided for each. A bit in the chunk header describes whether the objects contained in the chunk are atomic or composite. Atomic objects are marked and possibly released by the collector, but are never searched for embedded references. Composite objects are cleared when they are reallocated to prevent a subsequent mark phase of the collector from following obsolete references. Atomic objects are not initialized as a part of allocation.

We take explicit precautions against accidentally marking a free list as the result of a misidentified reference. In different versions of our collector, we either use an additional mark bit to identify objects on the free list, or we explicitly unmark objects on free lists that may have been accidentally marked.

Practical Experience

The garbage collection technique described above has been incorporated into the run-time system supporting the Russell implementation on Sun 3 workstations. In addition, variants have been used for a number of other applications. Most of these were originally written in C, and had previously been used without automatic storage management. Our experiences have been almost uniformly positive.

The Russell Collector

We use a stack-based non-recursive marking algorithm. The algorithm is structured so as to minimize stack growth for long, but shallow, linked lists. Since we are operating in a virtual memory environment without hard limits on the amount of available memory, stack size has not been a problem. The Schorr-Waite-Deutsch[21] link-reversal algorithm could have been used instead.

In the context of Russell programs, it suffices to mark only objects accessible through registers, or through pointers aligned on 32-bit boundaries in the stack. The versions of the collector described in the next section also start marking from the static data area, with fewer restrictions on alignment.

The allocator coexists with the Sun UNIX **malloc** utility. The latter 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.

Among the largest application programs we have run in this environment is an interpreter for Fortran intermediate code supporting constructive (exact) real arithmetic[7]. The interpreter consists of about 2500 lines of Russell code, plus a small amount of C code. This excludes the allocator and another 5000 lines of C code constituting the Russell run-time environment. It performs a large amount of heap allocation, both for arithmetic on large integers, and to support the heap allocated closures and activation records necessary to support functions as data objects. The following discussion is based on this interpreter running on a Sun 3/260 (25 MHz 68020), with a variety of input programs to the interpreter. The total heap size was generally about 2 megabytes.

The sweep phase of the garbage collector was measured to take about 0.4 seconds per megabyte in the heap. The mark phase took about 1.9 seconds per megabyte of *accessible* memory in the heap. Overall garbage collection times in this environment normally range from under 1 to around 3 seconds, depending on the Fortran program being interpreted.

For this application, approximately 2/3 of the bytes marked by the marking algorithm were composite, and most objects were very small, typically on the order of 10 bytes. Larger object sizes reduce mark times substantially. For a contrived program that preallocates a 1 Megabyte singly linked list consisting of 40-byte objects in a 2.2 Megabyte heap and then repeatedly forces garbage collections by repeatedly allocating 40 byte composite objects containing no valid pointers, the time taken by the mark phase changed to just under 1 second per megabyte of accessible memory.

In retrospect, our allocation strategy probably distinguishes between too many different object sizes. As a result we observed appreciable fragmentation. We maintain two different free lists (atomic and composite) for every object size that is a multiple of 4 bytes and less than 2K bytes. Though the resulting performance was always acceptable in the long run, we occasionally saw temporary situations in which the collector failed to reclaim nearly one half of available memory, because the corresponding objects were already on free lists corresponding to temporarily undesirable object sizes. The memory chunks in which these objects resided could not be reclaimed because they contained other objects that were still in use.

This resulted in shorter intervals between garbage collection, and slightly increased the observed mark times. This situation was noticeable only when the user program explicitly increased the size of the heap, thus allowing it to run for a long period without any reclamation of unused memory.

In general, garbage collection times are likely to remain reasonable, even for interactive applications, provided the total amount of memory in a particular address space is not excessive. Conventional Lisp or Smalltalk systems could exhibit unacceptable pauses, since the entire programming environment normally shares an address space. Under these conditions, a precise, generation-based collector would be more appropriate[27].

It is worth noting that the garbage collection time is more sensitive to the amount of *accessible* memory, than the amount of *available* memory. Large amounts of available memory decrease total garbage collection time, with a relatively small increase of garbage collection pauses. Additional *available* memory is usually helpful. However, unlike the case of a copying collector[4], huge increases may not be desirable.

It is tempting to avoid garbage collection pauses by running the collector in parallel with the user program. It does appear feasible to overlap the sweep phase of the collector with continued execution of the user code. In contrast, truly parallel[24,15] or interleaved[4] execution of the mark phase appears extremely difficult without substantial cooperation from the user code. A partial solution would be to conceptually checkpoint the heap at the beginning of the mark phase, and then to garbage collect the checkpoint. As is pointed out in reference [1], the checkpointing process can be reasonably implemented by copying pages of the heap only when a write access is attempted by the user process. One would hopefully avoid copying most of the heap as a result.

We have implemented a more primitive technique for avoiding *perceptible* garbage collection pauses. It is possible for the user program to voluntarily start a garbage collection when it expects to be idle for a few seconds, typically just before waiting for keyboard input. Such a voluntary garbage collection is immediately aborted if input becomes available. An abort during the mark phase simply discards information gathered so far. Aborting during the sweep phase results in partial reclamation of inaccessible objects.

Garbage Collection with Traditional Languages and Existing Programs

A collector similar to the one described in the previous section can be used with code produced by one of the standard UNIX compilers, provided that the user program does not encode pointers in clever ways, *e.g.* by adding tag bits to pointer values, and provided that no objects are accessible solely as a result of a pointer to the middle of the object. Both constraints are satisfied by typical programs.

The compiler should also not perform optimizations that introduce a violation of either constraint. Most common C compilers do not. A highly optimizing compiler might fail to preserve a ''dead'' reference to the beginning of an object. In this case, the collector should be modified to consider pointers to the middle of objects, as discussed in the preceding section.

Such a collector was successfully used in the run-time system for student-built compilers on a VAX.

Similar collectors were incorporated into two different code generators for the Russell compiler. Both were written in C and initially designed to use the standard allocation facility. One was converted in response to a storage allocation bug that proved difficult to trace. The other was converted primarily because, in the presence of explicit deallocation, and thus infrequent collections, the collecting allocator exhibited better performance. Both use explicit deallocation where this is convenient. The collector makes it possible to avoid it in subtle cases.

We used our approach to add garbage collection to two large existing systems, TimberWolf[22,23] and SDI. TimberWolf is a VLSI layout program consisting of 18371 lines of C code. It uses dynamic storage allocation via **malloc**, **realloc**, and **free** to construct complex linked lists representing the VLSI routing. 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. SDI was written in part to exercise most features in the Sun Microsystems SunView[26] window system. Approximately 34,000 lines of code in the SunView window system are tested by SDI. SunView again performs large amounts of allocation and deallocation for window management.*

Neither TimberWolf nor SDI needed to be changed, or even recompiled, to run with our garbage collector. They were simply relinked so that calls to Unix allocation routines instead called our allocator. Once we eliminated all the bugs in our collector implementation, TimberWolf ran without problems. SunView presented some interesting challenges. One of these apparently arose from dynamically allocated memory that is subsequently remapped to refer to the frame buffer. Fortunately, the storage allocated for the frame buffer is distinguished by being allocated by the Unix **valloc** call, and so we were able to work around the problem by simply never freeing storage allocated by **valloc**.

A second problem resulted from Sun's ''notifier''. It obtains large blocks of storage from **malloc** and divides them up itself, to improve allocation speed. The notifier does not keep a pointer to the head of exactly three of these blocks, again requiring that the collector especially recognize them and not free them. Changing our collector so that internal pointers prevented reclamation would also have removed this problem.

*34,000 lines is an estimate, based on a figure of 17 bytes of executable object per line of code seen in other C programs, and noticing that about 587,000 bytes of executable object (called 'text' by Unix) are linked from SunView with SDI by the Sun linker before execution.

Working around these two problems only took one of us about two days, without access to the SunView source code. As discussed above, certain programming styles involving disguised pointers (*e.g.* by XOR'ing them or adding tags) will not work with our collection method. The TimberWolf and SDI/SunView examples indicate that perhaps such practices are not common, and when they do occur are easily worked around. Programmers who knew their storage would be garbage collected should experience even less trouble.

Michael Caplinger recently used a similar system to circumvent storage leaks in an existing large system written in C[9]. His garbage collector is based on the ''conservative'' approach described here. Since his application required a fast implementation of the UNIX **realloc** facility, he used a first-fit linear search allocator instead of our segregated storage approach. His experience was also generally positive, though, as expected, the first-fit allocator introduces a performance penalty (except in the case of **realloc).**

Garbage Collection as a Debugging tool

When working with the SunView code we developed a key tool which illustrates the utility of a collector, even for code which is intended to do its own explicit management via calls to **free**. The tool is an allocation-and-free tracer, which is invoked optionally as part of the collector at run-time. It works as follows.

When the tracer is enabled, every call to **malloc** (or equivalent, such as **valloc**, **calloc**, or **realloc**) records the names of the subroutines on the stack at the time of allocation.† This means that every dynamically allocated piece of storage can be traced to its point of allocation. In a system without a collector, it is the allocating code that bears responsibility either for freeing the storage or for seeing that it is handed to some other piece of code that will take that responsibility.

This 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''. That is, it should deallocate memory whenever it becomes unused.

Memory leaks are easily traced to the allocating routine as follows: Run the tracer as described above. When **free** calls are made, mark that storage as freed. (Actually freeing it is optional.) When the collector runs, it will identify some storage as being reclaimable by virtue of having no pointers to it. Any such storage that was never explicitly deallocated with a ''free'' call is a likely candidate for a storage leak. There are no pointers to the head of that object, so it will be difficult to free in the future, and yet is probably not in use (again, because there are no pointers to it). Of course, it could just be that the owner of the object has a pointer to its middle, but we did not see such a case during our SunView debugging. When running with the tracer we had the collector print the call-stack of every piece of storage which was unmarked by the collector (and so reclaimable) and yet had never been explicitly ''free''d. Sample output is given in figure 4. Except for the two exceptions mentioned above, all were true memory leaks. Storage

† This recording is done via a simple symbol table obtained from the program's executable file, and a stack walk. The tracer consists of just a few hundred lines of code (most of it borrowed from an existing **longjmp** implementation), and took one of us a day to implement.

leaks are notoriously hard to find, and yet are most damaging to the most important programs--those which are intended to run correctly for long periods of time. Having a collector available can be very useful even for debugging code which does not intend to use it.

```
Collecting an unfreed block of size 12 at location 728C0.
        called from '_strspl' called from '_setenv'
        called from '_tool_parse_one' called from '_tool_parse_all'
        called from '_frame_set' called from 'window_set.o'
        called from 'window.o' called from 'gcsdi'
        called from '_doit' called from '_callCommand'
        called from '_commandloop' called from '_main'

Collecting an unfreed block of size 12 at location 1B1440.
        called from '_cursor_create' called from '_build_playing_fields'
        called from 'gcsdi' called from '_doit'
        called from '_callCommand' called from '_commandloop'
        called from '_main'
```

Figure 4: Output from the Allocation/Free Tracer during a SunView application

The Impact of "Conservatism"

In spite of careful observation, and in spite of testing of applications requiring a large number of garbage collection cycles, we have never witnessed any evidence that memory was lost as a result of the conservative collection strategy. In particular memory consumption was never significantly different from a VAX-based Russell implementation using a completely conventional allocator.

In order to gain some insight into the possibility of storage loss, we modified the marking procedure to report "near misses", that is, references to 4 byte boundaries in the address range occupied by heap blocks that did not correspond to valid object addresses. We discovered that such references were, in fact, exceedingly rare. We typically saw 4 to 5 per garbage collection of a 2 megabyte address space. (Of course this number can increase dramatically when the collector coexists with a foreign allocator, such as UNIX "malloc", since all references to "malloc" allocated data will be included in this count.) We have only found a "reference" to an object on a free list in a case in which a library routine had explicitly freed an object, but had neglected to explicitly clear the variable pointing to it. This could easily have been avoided by more careful coding of the library routine.

We examined several of the near misses in detail, and discovered that they all represented either pointers to the middle of otherwise accessible objects (most commonly activation records) or, occasionally, pointers created by the storage allocator itself and not cleared by another routine reusing its stack area. (In our case, the latter were references to the beginning of chunks. This phenomenon would probably never have been noticed if the final code generator for Russell had been a bit more mature. At the time it reserved excessive stack space for temporaries. This space was not immediately cleared.)

The first kind of misidentified reference is inherently benign. The second is rare and unlikely to cause long term problems, since presumably old stack contents will *eventually* be cleared. Thus it appears even less likely that noticeable unnecessary retention (beyond that exhibited by a conventional collector) would ever occur in practice.

Acknowledgements

This research was supported in part by NSF grant DCR-8607200.

Major pieces of the allocator and garbage collector were borrowed from a more conventional allocator originally written by Alan Demers. The Russell code generators mentioned above were written by Preston Briggs and Kumar Srikantan. Kumar Srikantan also added the facility for aborting garbage collections.

We would like to thank Michael Caplinger for sharing his experiences with us. We would also like to thank Bernard Lang, David Chase, and the referees for their comments on earlier drafts of this paper.

References

1. Abraham, S., and J. Patel, ‘‘Parallel Garbage Collection on a Virtual Memory System’’, Proceedings of the 1987 International Conference on Parallel Processing, pp. 243-246.

2. Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, p. 71.

3. Aho, A. R. Sethi, and J. Ullman, *Compilers: Principles Techniques and Tools*, Addison-Wesley, 1986.

4. Baker, Henry G., Jr., ‘‘List Processing in Real Time on a Serial Computer’’, *Communications of the ACM 21*, 4 (April 1978), pp. 280-294.

5. Boehm, Hans-J., Alan J. Demers, and James E. Donahue, ‘‘A Programmer’s Introduction to Russell’’, Technical Report 85-16, Computer Science, Rice University, 1985.

6. Boehm, Hans-J., and Alan Demers, ‘‘Implementing Russell’’, *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction, SIGPLAN Notices 21*, 7 (July 1986), pp. 186-195.

7. Boehm, Hans-J., ‘‘Constructive Real Interpretation of Numerical Programs’’, *Proceedings of the ACM SIGPLAN ’87 Symposium on Interpreters and Interpretive Techniques*, pp. 214-221.

8. Branquart, P. and J. Lewi, ‘‘A Scheme of Storage Allocation and Garbage Collection for Algol 68’’, in *Algol 68 Implementation*, J. E. L. Peck, editor, North Holland, 1971, pp. 199-232.

9. Caplinger, Michael, ‘‘A Memory Allocator with Garbage Collection for C’’, Proceedings of the Winter 1988 Usenix Conference.

10. Cardelli, Luca, ‘‘Compiling a Functional Language’’, *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp. 208-217.

11. Chase, David R., ‘‘Garbage Collection and Other Optimizations’’, Ph.D. Dissertation, Rice University, Houston, August 1987.

12. Christopher, T.W., ‘‘Reference Count Garbage Collection’’, *Software Practice and Experience 14*, 6 (June 1984), pp. 503-507.

13. Cohen, Jacques. ‘‘Garbage Collection of Linked Data Structures’’, *ACM Computing Surveys 13*, 3 (Sept. 1981), pp. 341-367.

14. Collins, G.E., ‘‘A Method of Overlapping and Erasure of Lists’’, *Communications of the ACM 3*, 12 (December 1960), pp. 655-657.

15. Dijkstra, Edsger W, Leslie Lamport, A. J. Martin, C. S. Scholten, E. M. F. Steffens, ‘‘On-the-Fly Garbage Collection: An Exercise in Cooperation’’, *Communications of the ACM 21*, 11 (Nov. 1978), pp. 966-975.

16. Donahue, J., and A. Demers, ‘‘Data Types are Values’’, *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), pp. 426-445.

17. Hanson, David R., ‘‘A Portable Storage Management System for the Icon Programming Language’’, *Software Practice & Experience 10*, 6 (June 1980), pp. 489-500.

18. Milner, R. ‘‘A Theory of Type Polymorphism in Programming’’, *Journal of Computer and System Sciences 17* (1978), pp. 348-375.

19. Rovner, Paul, ‘‘On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language’’, Report CSL-84-7, Xerox Palo Alto Research Center.

20. Ruggieri, Cristina, and Thomas Murtagh, ‘‘Lifetime Analysis of Dynamically Allocated Objects’’, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, (Jan. 1988), pp. 285-293.

21. Schorr, H. and W. Waite, ‘‘An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures’’, *Communications of the ACM 10*, 8 (Aug. 1967), pp. 481-492.

22. Sechen, C. and A. Sangiovanni-Vincentelli, ‘‘The TimberWolf Placement and Routing Package’’, *Journal of Solid State Circuits 20*, 2 (April 1985).

23. Sechen, C. and A. Sangiovanni-Vincentelli. ‘‘TimberWolf 3.2: A new Standard Cell Placement and Global Routing Package’’, *Proceedings of the 1986 Design Automation Conference*, Las Vegas, Nevada, June 29 - July 2, 1986, pp. 432-439.

24. Steele, Guy. L. Jr., ‘‘Multiprocessing Compactifying Garbage Collection’’, *Communications of the ACM 18*, 9 (Sept. 1975), pp. 495-508.

25. Steele, Guy L. Jr., ‘‘RABBIT: A Compiler for Scheme’’, AI Memo 474, Massachusetts Institute of Technology, May 1978.

26. Sun Microsystems, Inc., *SunView Programmer’s Guide*, September 1986.

27. Ungar, David, ‘‘Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm’’, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices 19*, 5 (May 1984), pp. 157-167.