Simple Generational Garbage Collection and Fast Allocation

Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544

March 1988
revised September 1988

ABSTRACT

Generational garbage collection algorithms achieve efficiency because newer records point to older records; the only way an older record can point to a newer record is by a **store** operation to a previously-created record, and such operations are rare in many languages. A garbage collector that concentrates just on recently allocated records can take advantage of this fact.

Such a garbage collector can be so efficient that the allocation of records costs more than their disposal. A scheme for quick record allocation attacks this bottleneck.

Many garbage-collected environments don't know when to ask the operating system for more memory. A robust heuristic solves this problem.

This paper presents a simple, efficient, low-overhead version of generational garbage collection with fast allocation, suitable for implementation in a Unix environment.

Keywords: garbage collection, dynamic memory allocation

Introduction

Many programming environments have heap storage management with automatic garbage collection. In these systems, storage that is no longer accessible through chains of pointers from global variables or the runtime stack is automatically re-used. (The global variables and the runtime stack will be called the set of root pointers.) There are two old and simple algorithms for garbage collection, upon which many improvements have been based.

The ‘‘mark-and-sweep’’ algorithm[1] first traverses and marks— using depth-first search — all of the records reachable from root pointers. Then all the records in memory are examined in order of their addresses, and the unmarked records are put onto a free list, to be re-used when the user's program (called the *mutator**) allocates new records.

The ‘‘copying’’ algorithm[2] traverses the records reachable from root pointers, copying these records to another area of memory (the destination space) unused by the mutator. Any pointers to these records will be replaced by pointers to the copies in the destination space. When the collector is finished, the destination space will contain a logical copy of only the reachable records of the mutator's space. These records will be contiguous in the destination space, even though they are scattered (separated by garbage records) in the mutator's space. All root pointers now point to the destination space. At this point, the destination space and the mutator's space are exchanged. Copying algorithms do not have a ‘‘free list:’’ there is always a contiguous area of free memory at the end of the mutator's space, and records can be allocated from the beginning of this area. (When this area becomes exhausted, of course, the garbage collector must be invoked.)

The copying algorithm has the advantage that it takes time proportional only to the number of reachable records, no matter how much garbage there is, while the mark-and-sweep algorithm takes time proportional to the size of memory (the sum of the numbers of reachable records and garbage records). As memories get larger, the ratio of reachable records to garbage decreases, so that copying algorithms perform more efficiently than mark-and-sweep algorithms[3].

In some programming languages, when a record is allocated it is immediately initialized (as part of the allocation procedure) to point to other records, and the contents of existing records are only rarely altered. Two observations can be made about such languages[4] that lead to an improvement of the copying garbage collection algorithm:

1. The graph of records and pointers is mostly acyclic: newer records point to older records, and older records do not point to newer records. (When a record is created, it is initialized to point to already-existing (i.e. older) records.) An older record can point to a newer record only if the old record is altered after it is initialized; and such *assignment* operations are rare in these languages. In this paper, an ‘‘assignment’’ will mean the changing of a field in a previously allocated record, not the initialization of a new record.
2. A young object is more likely to become garbage soon. Most objects have short lifetimes, and only a few objects have long lifetimes. If something has lasted for a long time already, then it's probably part of a semi-permanent data structure; if the object is relatively new, it's likely to be a temporary structure holding an intermediate result.

---

*In the perverse jargon of garbage collection, the user's process is called the *mutator* because it is busy changing all the data that the garbage collector has just cleaned up.

Generational garbage collection[4, 5] is a variant of copying garbage collection that takes advantage of these observations. Records are kept in several areas A_i of memory; records in the same area are of similar age, and all the records in area A_i are older than the records in area A_{i+1}. This implies (by observation 1 above) that for $i < j$, there are (almost) no pointers from area i into area j (see Figure 1.)

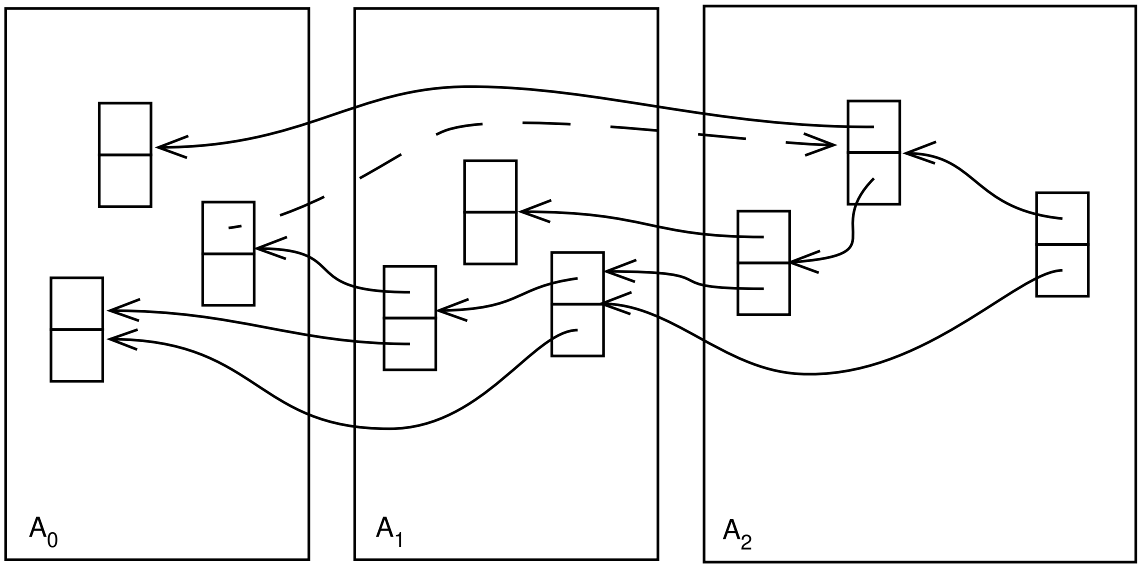

Figure 1: All the records in area A_2 are newer than the records in A_1; the oldest records are in A_0. The pointer represented by the dashed line is anomalous; it must have been created by an assignment that modified an old object.

Consider the newest area, A_n. The copying garbage collector may be applied to this area, using the global variables and runtime stack as a root set. Because (almost) no pointers in the other areas can possibly point to records in A_n, the other areas (almost) need not be considered as roots. The garbage collector may copy the reachable objects of A_n into a new area $A_n{}'$ without touching the other areas. This is advantageous because A_n is expected (by observation 2 above) to have a higher proportion of garbage than any of the other areas.

In general, the garbage collector can work on area A_i if it also works on all the newer areas; that is, it may work on any subset $A_{i,} A_{i+1}, \ldots, A_n$ of the areas. For greatest efficiency, the garbage collector should usually copy just A_n, but occasionally copy the older areas.

This algorithm depends on the fact that records contain pointers only to older records. In the presence of assignments to fields of old records, this acyclic condition will not hold. Lieberman and Hewitt[4] handle this problem by making such fields point indirectly through an ‘‘assignment table.’’ If a record in area A_i has been assigned into, then that field — instead of pointing directly to its referent — points to an element of a table associated with area A_i. That element of the table contains the assigned value. When area A_n is collected, all of the roots in the older areas can be found in the assignment tables associated with those areas.

This method has two disadvantages. Any fetch from a stored field must be done indirectly. That is, whenever a field of any record is fetched, it must be examined to see if it points to an element of an assignment table. If so, it must be dereferenced again. Checking every value fetched is very expensive without special hardware.

The second disadvantage of this method of handling assignments is that every time A_n is collected, all of the assignment tables must be examined. These assignment tables can be quite large, as they contain information about all of the (reachable) records ever assigned into — not just one ones that point into the new space. The overhead of searching the assignment tables for pointers into A_n can be greater than the time spent copying A_n. Even though we make the assumption that assignment operations are rare, this does not prevent the cumulative number of assignments from becoming large.

The overhead of searching the assignment tables can be ameliorated by Moon's ''ephemeral garbage collection'' algorithm[6]. This is a variant of the Lieberman-Hewitt scheme that manages the assignments differently. It uses special hardware in the virtual memory manager to detect all assignments of references to ''ephemeral'' objects — objects in area A_n (or A_{n-k}, for small k). Whenever an assignment is done, this special hardware examines the value stored, and the location stored into. If the value stored is a pointer into an ephemeral space, then the page containing the location stored into is marked in a special hardware table. Then when a garbage collection is done of space A_n, only the marked pages need be examined for roots. No store-table or extra indirection is used; fetch operations are therefore not affected.

With this form of ephemeral garbage collection, the overhead when collecting just area A_n is proportional to the number of assigned records that are really roots into A_n. In the original Lieberman-Hewitt scheme, all records ever stored into are examined. The use of low-overhead ephemeral garbage collection produces a noticeable improvement in garbage-collector efficiency[6].

Ungar's method[5] is to keep a list or vector of all the old objects that point to new objects. Whenever an assignment is executed, the compiled code checks to see whether the assigned-into object is old and the assigned value is a pointer to a new object. If so, the address of the assigned-into object is added to a list of root pointers. This list must be examined on each garbage collection.

If assignments are very frequent, then the root list will be large, and the cost of maintaining it will outweigh the benefits of generational garbage collection. Therefore, generational algorithms are probably not appropriate for Algol-like languages in which assignment is the principal mechanism for manipulating data structures. However, for mostly-functional languages like Lisp, ML, and Prolog, generational garbage collection is certainly worthwhile.

Both the original Lieberman-Hewitt scheme and Moon’s improvement are *incremental* — the mutator is never interrupted for more than a bounded amount of time — and require special hardware for their efficient implementation. Ungar’s method is not incremental, but requires no special hardware. Most of the fastest and cheapest computers available today, however, are general-purpose computers without any special garbage-collection hardware, so Ungar’s method is preferable in most circumstances.

There is an incremental algorithm that doesn’t require special hardware[7], but it requires more sophisticated operating systems than are commonly available. Therefore, it won’t be described in the present paper, which is meant to provide a easy-to-implement ‘‘recipe’’ for efficient and portable garbage collection.

The copying algorithm

The heart of a generational garbage collector is a mechanism to copy the live data of one region of memory to a contiguous group of records in another region. This can be done without an auxiliary stack or queue; Cheney’s algorithm[8] uses the destination region as the queue of a breadth-first search:

We start with a set of ‘‘root pointers.’’ Any record in the ‘‘source region’’ that is reachable from a root pointer will be copied. In the ‘‘destination region’’ there are two pointers, *scan* and *next* that are initially at the begining of the region.

For each root pointer *R*, we perform the following procedure, replacing *R* by *forward*(*R*):

```
forward(R) =
 if R points into the source region
   then if R[1] points into the destination region
          then return R[1]
          else copy the record pointed to by R to location NEXT
               assign NEXT into R[1]
               increment NEXT to point past the copy of the record
               return R[1]
   else return R
```

In this program, R[1] means the first field of the record pointed to by R.

The forward procedure copies the source record to the destination, and returns a pointer to the destination; unless the record has been previously copied, in which case R[1] points to the copy, and R[1] is returned without making a new copy.

After the procedure *forward* is applied to each root, it is then applied to each word in the destination region. The *scan* pointer is successively incremented through the destination region, and *forward* is applied to each word it points to. Of course, this may cause more records to be copied, so that *next* will also be incremented in this phase. When *scan* catches up with *next*, the algorithm is finished.

Some environments contain objects (e.g. character strings) that contain no pointers, but that may contain data that could be mistaken for pointers. If it is arranged that such objects are identifiable by the format of their first word, then whenever the *scan* pointer reaches such an object, the pointer should be made to ‘‘skip’’ to the end of the object without forwarding its contents.

Fast allocation

With copying or generational garbage collection, there is no lower bound on the cost of freeing a record by garbage collection[3], primarily because the garbage collector never touches the garbage (just the live records). The cost of garbage-collecting a record can be less than one instruction, on an amortized basis.

In that case, it is worthwhile ensuring that allocating a record is also cheap. Assume (temporarily) that each cell holds two pointers. In LISP, an allocation is typically expressed as **(cons A B),** meaning ‘‘allocate a two-word record containing the values **A** and **B,** and return a pointer to it.’’ Let us assume that in any scheme, the values **A** and **B** must be stored into memory; any instructions other than the ones to store **A** and **B** count as overhead.

With a compacting garbage collector, the unallocated memory is always a contiguous region. That is, there is no ‘‘free list;’’ instead, there is a free area of memory. The function **(cons A B)** could be implemented with these machine instructions:

1. Test free-space pointer against free-space limit.
2. If the limit has been reached, call the garbage collector.
3. Subtract 2 (the size of a cons cell) from the free-space pointer.
4. Store A into new record.
5. Store B into new record.
6. Return current value of free-space pointer.

This code sequence assumes that the records are allocated starting at the higher addresses and moving towards the lower addresses.

We can use the virtual memory hardware of the computer to accomplish the test in line 1. If an inaccessible page is mapped to the region just before the free space, then any attempt to store there (in line 4) will cause a page fault. This trap can be returned to the run-time system, which will initiate a garbage-collection.

The free-space pointer can be kept in a register to simplify access to it. Furthermore, on a VAX the subtraction from the free-space pointer can be implemented by means of an auto-decrement addressing mode. The new instruction sequence for **(cons A B)** is thus:

1. movl B,-(fsp)
2. movl A,-(fsp)

At this point, the pointer to the new record is available in the free-space pointer (fsp) register for appropriate use. Because these two instructions take no more time than any other pair of stores into memory, the overhead attributable to an allocation from the heap is exactly zero.

Variable-sized records

In pure LISP all records are the same size, but in most languages records are of various sizes. The garbage collector must determine the size of each record that it copies; there are various schemes for accomplishing this:

1. Each record can contain a tag word at the beginning that explains the format of the record.
2. Records of different formats may be stored in different areas of memory; by examining the address of the record the garbage collector may look up its format.
3. In languages with static type systems (like Pascal or ML), the compiler can provide the garbage collector with a map of the type system, which will explain the format of run-time data[9, 10].

In this paper, the record-tagging scheme will be used, as this simplifies the layout of records in memory.*

When there are variable-sized records, it is slightly more complicated to allocate records. In particular, it may not be as easy to use the auto-decrement mode in allocation (if the computer has one). This is because we would like the ‘‘out-of-space’’ page trap to occur at the very beginning of allocating a record; otherwise it is far too complicated to deal with a half-allocated record.

When all records are of size 2 (as in the previous section), then the page trap would always occur on the first store of the two because pages have even lengths. But when there are variable sizes, a record may cross a page boundary, so we must ensure in some other way that the first store causes the trap. What we can do is store the *last* word of the record first. If that store succeeds, then all the rest are guaranteed to (as long as the record is not larger than a page, or as long as there are many inaccessible pages in a row).

* The garbage collector must also be able to distinguish between pointers and non-pointers (e.g. integers). In Algol-like languages, the record-format tag can indicate which of the fields are pointer fields; in dynamically-typed languages (like Lisp), each field must contain its own indicator. Integers may be kept in an area of memory distinguished by address range, in which case the representation of an integer must be a pointer into this area; or integers may be distinguished from pointers by a different high- or low-order bit. For example, on byte-addressable machines, the low-order bit of all pointers is zero; so integers could be represented by 31 data bits and a low-order 1 bit. Alternatively, pointers can be made odd and the integer tag-bit can be 0, simplifying tagged-integer arithmetic.

Finally, (for reasons that will become clear in the next section), we will allocate records in order of increasing address. With all of these modifications, the VAX code for allocating a three word record initialized to (A,B,C) looks like this:

```
movl    C,12(fsp)     #  only this instruction can trap
movl    B,8(fsp)
movl    A,4(fsp)
movl    $3,0(fsp)    #  store the format descriptor
movl    fsp,dest     #  keep the pointer to the new record
addl2   $16,fsp
```

How to arrange the generations

Most implementations of generational garbage collection use several generations of varying ages. Using just two generations may be most efficient, because it maximizes the size of the newest space, and therefore reduces the likelihood that a young record will *ever* be copied. Whenever the ‘‘newer’’ region is copied, all the records in it can be promoted into the ‘‘older’’ region. Because young records have a high mortality rate, the newer region contains few live records, and copying the live data of the newer region onto the end of the older region is relatively fast; this will be called a ‘‘minor collection.’’

Eventually, after dozens of minor cycles, the older region fills up. At this point it is also mostly garbage, since the it has taken such a long time to fill that even the ‘‘mature’’ records are suffering from mortality. We can then do a copy of the older region’s live records, called a ‘‘major collection.’’

We would like to arrange the regions in memory so that an inaccessible page is at one end of the newer region; we will allocate records in the newer region until we trap on that page. Let us examine a map of the Unix address space to look for such a page (figure 2).

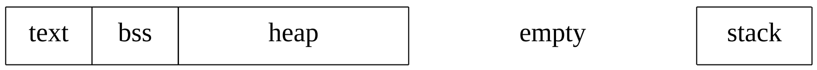

Figure 2: The Unix address space.

The pages in the empty region immediately below the stack are automatically created when stored into; and the stack cannot always be made large enough for a good-sized heap. The upper end of the stack region and the lower region of the heap/bss region are inconvenient to use because that would destroy static variables. The most convenient inaccessible page to use as the boundary of the newer region is at the upper end of the heap; this is known in Unix as the program break.

So we will use the heap area for the older and newer regions. The newer region must start somewhere in the middle and grow toward the upper end; when it tries to grow into the inaccessible page we will handle the segmentation-fault signal and initiate garbage collection.

Whenever we copy the live data from one region into another, we expect that the amount of data copied is much smaller than the source region; the vast majority of records are garbage and don't get copied. Nevertheless, we should design a system that will never die when it doesn't have to. So we will ensure that the destination region of any collection cycle is as large as the source region. In general, this means that if the program uses at most A words of live data, then the size of memory M must be at least $2A$. We will attempt to preserve the following invariant:

- If the amount of live data is less than half the size of the heap memory, then the garbage collector will never run out of space.

Of course, whenever A approaches $M/2$, then the performance of the garbage collector will seriously degrade, with frequent collections. For best performance, M should be significantly larger than $2A$.

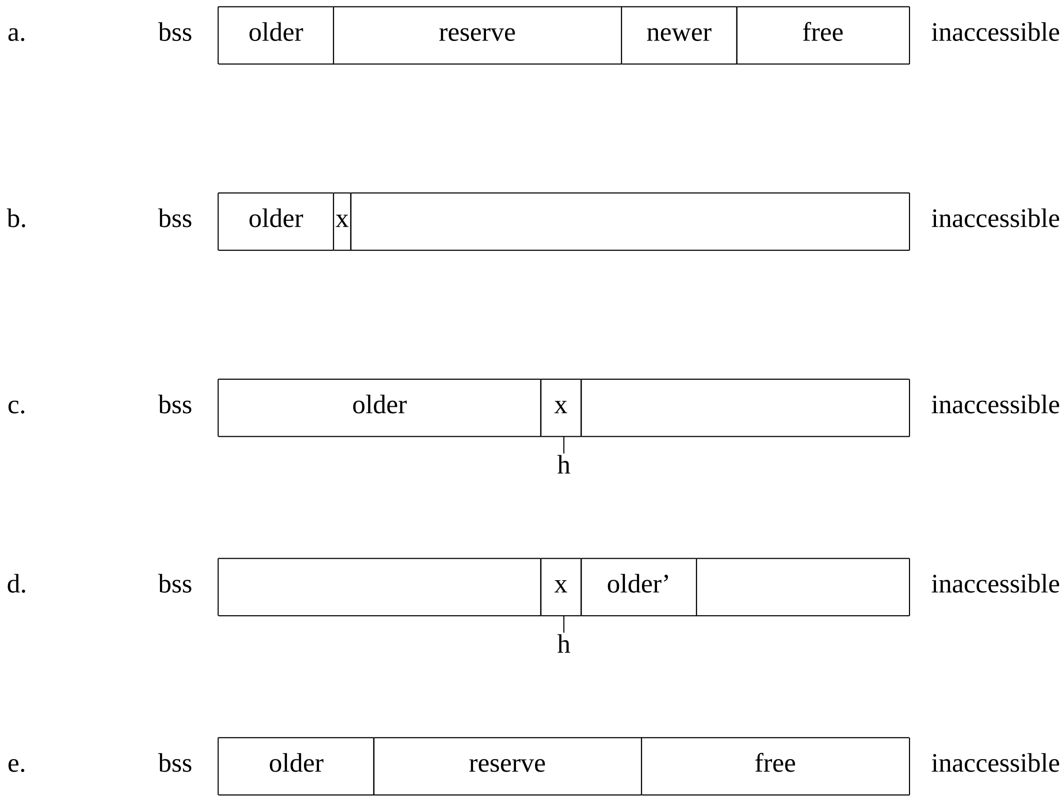

Figure 3: Layout of the heap.

The heap will be arranged as in figure 3a. The *newer* region grows into the *free* region until it hits the inaccessible page. At that point, the live data from the newer region is copied into the *reserve* space at the end of the *older* region, into the area marked *x* in figure 3b. Then, the empty region to the right of *x* must be divided in half to make the *reserve* and *free* regions, and allocation of new records may resume. Note that the boundary between *reserve* and *newer* has changed by an amount $x/2$ since the last cycle.

At a certain point, the older region must be examined: its live records must be copied. This must occur when the older region is about half the size of the heap area. Consider the situation immediately after a minor collection cycle (figure 3c). Here the point h marks the middle of the heap area, and a minor collection has just caused the x region to straddle this point. At this point we would like to copy the live data from the older region into the space to the right of x; how do we ensure there's enough room?

If we copy all of the live data in the regions $older + x$ into the reserve region, then reserve region still might not be large enough (though that is unlikely), because the reserve region to the right of x is not quite half the memory size. Remember, we undertake to guarantee that as long as the amount of live data A is less then half the memory size M, the garbage collector won't fail. However, we can copy all the live data from the older region into the reserve region, and leave x where it is* (figure 3d). The region $older'$ contains all of the copied live data from the older region. Since we know that all of the data in x is live (since there was just a minor collection that copied that data), then the amount of data in $older'$ is $A - |x|$, which is guaranteed to fit in the region to the right of x.

At this point, we can block-move the regions x and $older'$ to the beginning of the heap area to become the next *older* region, and then start another minor cycle (figure 3e).

When to ask for more memory

Let γ be the ratio of memory size to live data. When γ is less than 2, the algorithm will run out of space; when γ is near 2, its performance degrades. For good performance, γ should be 3 or more. In the Standard ML of New Jersey compiler[11], which makes extremely heavy use of the allocator, setting $\gamma = 3$ gives a garbage-collection overhead of about 11%, and $\gamma = 7$ reduces the overhead to about 6%. In environments that don't allocate so wantonly, $\gamma = 3$ should yield very low overhead (two or three percent). (The ML system's garbage collector can do a 1-megabyte major collection in about 2 seconds on a VAX-8650; a typical minor collection takes 40 milliseconds.)

We will define γ_0 as the desired value of γ. Of course, we can't always know in advance the maximum size of the live data A. When the program starts using too much live data for the given heap size, we should ask the operating system for more space. In Unix, this is done by the `brk()` system call, and it can be done just after a major collection under any of the following circumstances:

---

*The pointers in region x must be scanned, since they are roots of the older region.

1. After a major collection when $\gamma < \gamma_0$. We know that after a major collection, all of the data in the older region is live, so γ is easily calculated.

2. The collector runs out of space when trying to copy the older region into the reserve region to the right of x. This means that $\gamma < 2$, so more memory is unequivocably needed.*

3. After a minor collection cycle, the free region is not much larger than the amount requested by the mutator on the allocation that initiated collection. This means that the mutator is requesting a very large object that, when allocated, would probably increase A above M/γ_0.

In any of these circumstances, it is convenient for the garbage collector to extend the heap area to the right by a call to `brk()`, and immediately make use of the new space.

If an attempt is made to allocate a very large object, then (by case 3 above) more space will be requested; but the allocation might again fail because M was not increased sufficiently. This doesn't require a special case in the algorithm, since the M will be increased on each failure until the allocation succeeds.

In some environments it may be useful to have more than two generations. The arrangement described here, wherein the collector merges newly-copied cells into an older generation, is extensible to more generations. However, there is an annoyance: We want to ask the operating system for more memory only when live data grows to a certain size. But live data can only be measured when all the generations are collected together. As more generations are used, it will be increasingly inconvenient to ask for more memory only after a ‘‘most-major’’ collection, or to ask for more memory inaccurately.

Keeping track of assignments

When applying the copying garbage collector to the newer area the root set will be the runtime stack, global variables, registers, and the set of all records in older areas into which any pointer to the newer area has been assigned. In systems where procedure call frames are heap-allocated, then the frame-pointer register will be a root, and there is no runtime stack. This can be a considerable savings in a functional-language implementation where call-frames are never modified, as it could be that most of the records in the linked list of call frames are in older areas, and do not need to be examined as possible pointers into the newer area.

The mutator maintains the root set; the garbage collector uses the root set when it starts collecting the newest area. The set given to the garbage collector may be a superset of the root set, since the garbage collector can easily tell which elements of the set really point into the new space. This leads to a very simple algorithm† for maintaining the root set: make a list of every record assigned into.

---

* When there are pointers from the older region into the newer region (caused by assignments, see section 7), it may be that the collector can be made to ask for more memory even when A is less than $M/2$. Thanks to Steve Vegdahl for pointing this out.

† Maintaining the root set is simple when there are only two generations, as described in this paper; for several generations, there need to be several root sets.

This can be done in software, with help from the compiler. After every instruction of the form ‘‘assign p into record r’’, the compiler will insert an instruction of the form ‘‘add r to the list of assignment destinations.’’ This list need not be a linked list — a contiguous vector of addresses will suffice. If the vector overflows, then more space for it can be allocated (not necessarily contiguous to the rest of the vector); or the garbage-collector can be invoked. In the simple scheme described in the previous section for arranging the memory areas under Unix, a linked list is the simplest implementation; the records in this list can be allocated from the same free area as ordinary records used by the mutator.

Let P be the assigned value, let R be the memory cell assigned into, The assignment operation will look like this:

> Store P at address R
> Put a record containing the address R on the linked list.

Of course, this operation may fail because the free space is exhausted; the trap handler must complete the operation before invoking the garbage collector.

When the garbage collector is invoked, it must examine the list of assignment destinations (the root set), and process only the elements (R, P) meeting the following criteria:

> The value R is not in the space being collected
> The value P is in the space being collected
> The value R does not already appear in the root vector

If R is in the space being collected, it will be reached anyway if it is accessible; and if it is not accessible, it will be removed. If P is not in the space being collected, then it isn’t a root for this garbage collection.

Though the address R can appear many times in the root vector, only one of these occurrences need be considered. Actually, it does no harm to leave duplicate occurrences of R in the vector — the first time R is examined, record P will be copied to the destination space and the new value of P will be stored at R. The next time R is examined, the new P will be found there; and pointers into the destination space are recognizable as such.

If there are several generations, those elements of the assignment list that satisfy the criteria above must be retained in a pruned copy of the root vector for the next garbage collection. However, in the arrangement described in this paper the copied records of the newer region are merged with the older region. Therefore, the assignment list may be discarded after each minor cycle. This is a great convenience.

With this understanding of how the garbage collector processes the root vector, we can calculate the true overhead per store operation. There are about four instructions of overhead inserted by the compiler, as described above; but the tests that the garbage collector makes must also be counted as overhead. To test whether R is in the older area takes one comparison (i.e. compare the address R with the highest address of the older area). To test whether P is in newer area also takes just one comparison. Thus, the overhead per store operation is approximately:

4 instructions to update the root list
4 instructions to examine a record on the root list

Store operations into previously allocated records are rare. The static frequency of store operations in a typical Lisp system is about 3%[6] and in the author's Standard ML compiler (written in Standard ML) the dynamic frequency of pointer stores is less than 1%. The algorithm described here increases the cost of a store by a factor of about 10; thus the system as a whole (including that part of the garbage collector that processes the root vector) should suffer an increased overhead of about 5% to 30%.

Handling page traps

When the mutator runs out of space, a page trap occurs: the allocator has tried to make a new record that extends into an inaccessible page. In Unix, this is a segmentation fault, and can be handled by a user routine; this routine must invoke the garbage collector, reset the free-space-pointer register to an appropriate value, and return from the interrupt.

When the garbage collector is invoked, the following pointers serve as roots to the newer region:

1. Global variables
2. The run-time stack
3. The machine registers
4. The assignment list

It is easy to find the global variables*, but how are the registers to be manipulated? (The head of the assignment list is generally kept in a register or a global variable.)

When Unix handles a segmentation fault signal, the kernel pushes some of the user registers onto the run-time stack and transfers control to the user function. When the user function returns, the kernel pops the registers from the stack.

If some of these registers point to data in the newer region, then as the data is moved to the older region the registers must be modified. It is simple enough to modify the copies on the stack and let the kernel copy the modified values back to the original registers.

* In the run-time system for Standard ML of New Jersey[11] it is particularly easy to handle global variables and the run-time stack, because neither exists! All records are accessible from registers.

Some registers are not pushed by the kernel, but it is simple enough for the user's function to push them and pop them.

At this point one opens the Unix manual to see in which order the kernel pushes the registers. Unfortunately, this order is not documented. Furthermore, the set of registers pushed differs in various versions of Unix. To find out the order on your machine, you can go into the machine-language debugger, put known values into registers, and then cause a segmentation trap to be handled. In the handling function, you can use the debugger to examine the stack to see where your known values ended up.

Programs that rely on such information learned in the debugger are bound to be system-dependent. Instead, we can have our run-time system do the same experiment each time it begins execution. Known values are loaded into registers, a page-trap is generated, and the handling function examines the stack to see where the known values are. (Beware: sometimes the same register is pushed more than once!) From this information, a register location map is made; subsequent traps can use this map to tell the garbage collector about registers.

Let us consider an example. A record of size two is to be created, initialized to the values currently held in registers 3 and 5. A pointer to the new record is to be put into register 2. The free-space-pointer is kept in register 12. The head of the assignment-list is kept in register 11. Registers 0 through 6 can contain local variables and temporaries that may be pointers into the garbage-collectible region:

```
movl   r5,8(r12)
movl   r3,4(r12)
movl   $2,0(r12)
movl   r12,r2
addl2  $12,r12
```

Normally, this instruction sequence would proceed at full speed. But this time, there aren't three words left in the free region, and the first instruction traps.

The kernel pushes the program counter, stack pointer, and registers 0, 1, and 2. The user's trap-handler pushes the rest of the registers. The trap-handler passes the vector of locations of the pushed registers to the garbage collector as roots, and notes specially the stack pointer and assignment list.

The garbage collector modifies the memory locations containing the saved registers as the records they point to are copied. The collector also modifies the saved free-space-pointer (r12) to point to the new beginning of the free-space region, and it clears the assignment-list pointer.

Finally, the trap-handler returns. The saved registers are popped back into the machine registers, and the mutator resumes at the beginning of the instruction that trapped.* This time it succeeds.

Cormack[12] describes another good scheme for access to registers by user processes.

Conclusion

Generational garbage collection is a great thing for languages that don't have too many assignments. The garbage collector becomes so fast that the overhead of creating records can dominate the overhead of reclaiming garbage, so it makes sense to worry about fast methods of creating new records. This paper has presented a simple, efficient arrangement of a generational garbage collector, and has described a way of chopping several instructions out of the allocation overhead.

In the runtime system for Standard ML of New Jersey[11] this garbage collector is a C program of about 500 lines. Although the compiled code allocates new records like crazy (about one word of newly-allocated memory for every 30 instructions executed; most becomes garbage), the overhead of garbage collection is still only about 5 to 10% (depending on the ratio γ_0 of memory to data).

Acknowledgements

Thanks to Trevor Jim and David MacQueen for many useful discussions, for helping with the debugging, and for comments on the manuscript.

References

1. John McCarthy, ''Recursive functions of symbolic expressions and their computation by machine - I,'' *Communications of the ACM,* vol. 3, no. 1, pp. 184-195, ACM, 1960.

2. Robert R. Fenichel and Jerome C. Yochelson, ''A LISP garbage-collector for virtual-memory computer systems,'' *Communications of the ACM*, vol. 12, no. 11, pp. 611-612, ACM, 1969.

3. A. W. Appel, ''Garbage collection can be faster than stack allocation,'' *Information Processing Letters*, vol. 25, no. 4, pp. 275-279, 1987.

4. Henry Lieberman and Carl Hewitt, ''A real-time garbage collector based on the lifetimes of objects,'' *Communications of the ACM*, vol. 23, no. 6, pp. 419-429, ACM, 1983.

5. David Ungar, ''Generation scavenging: a non-disruptive high performance storage reclamation algorithm,'' *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments)*, vol. 19, no. 5, pp. 157-167, ACM, 1984.

*On some machines (like the MC68020), the program counter saved by the kernel won't be pointing to the beginning of the trapping instruction, and the trap-handler routine must adjust the program counter.

6. David A. Moon, ‘‘Garbage collection in a large LISP system,’’ *ACM Symposium on LISP and Functional Programming*, pp. 235-246, ACM, 1984.

7. Andrew W. Appel, John R. Ellis, and Kai Li, ‘‘Real-time concurrent collection on stock multiprocessors,’’ *Proc. ACM SIGPLAN ’88 Conf. on Prog. Lang. Design & Implementation,*, pp. 11-20, ACM, June 1988.

8. C. J. Cheney, ‘‘A nonrecursive list compacting algorithm,’’ *Comm. ACM*, vol. 13, no. 11, pp. 677-678, 1970.

9. Dianne E. Britton, ‘‘Heap Storage Management for the Programming Language Pascal,’’ Masters Thesis, University of Arizona, 1975.

10. Andrew W. Appel, ‘‘Runtime Tags Aren’t Necessary,’’ CS-TR-142-88, Princeton University, 1988.

11. Andrew W. Appel and David B. MacQueen, ‘‘A Standard ML compiler,’’ in *Functional Programming Languages and Computer Architecture (LNCS 274)*, pp. 301-324, Springer-Verlag, 1987.

12. G. V. Cormack, ‘‘A micro-kernel for concurrency in C,’’ *Software — Practice/Experience*, vol. 18, no. 5, pp. 485-492.