Garbage Collection Can Be Faster Than Stack Allocation

Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544

June 30, 1986
Revised Jan 15, 1987

ABSTRACT

A very old and simple algorithm for garbage collection gives very good results when the physical memory is much larger than the number of reachable cells. In fact, the overhead associated with allocating and collecting cells from the heap can be reduced to less than one instruction per cell by increasing the size of physical memory. Special hardware, intricate garbage-collection algorithms, and fancy compiler analysis become unnecessary.

1. Introduction

Many modern programming environments use heap storage with garbage collection. Allocated cells that are not reachable (via a chain of pointers) from the runtime stack are the ''garbage,'' and they are ''collected'' for re-use by a traversal algorithm. This algorithm typically marks all of the reachable cells using a depth-first search, and then collects all the unmarked cells[1]; this takes time proportional to total number of (reachable and garbage) cells.

In languages without garbage collection (i.e. Pascal, C, etc.), the programmer must write bookkeeping code to keep track of heap-allocated cells, and free them explicitly when they are no longer needed. This can make programs significantly more complex.

In languages with garbage collection (LISP, Mesa, Icon, ML, Mainsail, etc.) the programmer need not worry about the bookkeeping of allocated cells; this makes programs simpler and more straightforward. On the other hand, because the garbage collector is often slow and expensive, these simpler and more straightforward programs are typically less efficient because they use garbage collection.

One trend in optimizing compilers for garbage-collected languages is to have the compiler deduce (statically) which cells may be freed[2]. This approach is a good compromise — some of the cells will be freed automatically, which reduces the load on the garbage collector; but the complexity is in the optimizing compiler, and not in the compiled program.

This paper shows that, with enough memory on the computer, it is more expensive to explicitly free a cell than it is to leave it for the garbage collector — *even if the cost of freeing a cell is only a single machine instruction.*

2. Copying garbage collection

The traditional mark-and-sweep algorithm, which puts all free cells onto a linked list for later re-use, takes time proportional to the number of reachable cells plus the number of garbage cells. For the purposes of this paper, however, we need an algorithm with a time complexity independent of the number of garbage cells.

In its simplest form, a *copying* garbage collector[3] works in two equal-sized memory spaces, only one of which is in use at a time. When it is time to collect, the garbage collector traverses all of the

reachable cells in the active space, and copies them into the inactive space. Then the spaces are switched; that is, the other space is now used, and the formerly active space is left empty until the next garbage collection.

The traversal of reachable cells can be done using a depth-first search, which takes time proportional to the number of reachable cells. The copying takes a constant overhead per cell, in addition to some time proportional to the total size of the cells copied.

A most important property of the copying garbage collection algorithm is that it never visits a garbage cell, so that the execution time of the garbage collector is dependent only on the number (and size) of reachable cells, and is independent of the amount of garbage.

The cost of one garbage collection may be computed as follows: Let A be the number of reachable cells, and let s be the average size of each cell. Let M be the size of each of the two memory spaces. The depth-first traversal and copying requires a constant number of operations per cell, plus a constant number of operations per pointer: $(c_1 + c_2 s)A$. The garbage collection is just the depth-first traversal plus the copying; the time spent is independent of M.

Assuming the number of reachable cells is approximately the same at each garbage collection, we can compute the cost-per-cell of garbage collection. That is, we can calculate, for each cell, the amount of garbage collection overhead associated with it.

To do this, the number G of cells allocated between garbage collections is divided by the cost of garbage collection. (Since we are assuming that A does not change much, then G is also the number of garbage cells.) However, we can compute G from the parameters above, since we wait until the active space is full before collecting:

$$G = \frac{M}{s} - A$$

This is just the number of cells remaining in the active space after the ''permanent'' reachable cells are copied there.

Now we can compute the cost per cell of garbage collection:

$$g = \frac{(c_1 + c_2 s)A}{M/s - A}$$

A glance at this formula shows that g can be made smaller by making M bigger.

3. Explicit freeing is more expensive

In this section we show that, if there is enough memory available, then an optimizing compiler's effort in stack allocation actually makes the compiled programs run slower.

Let us suppose that the cost of popping a record from the stack, or of explicitly freeing a garbage cell, is some constant f. (In fact, the former operation is typically cheaper than the latter, but as long as each is a constant, then the analysis is similar.)*

Under what conditions is f — the cost of an explicit free operation — greater than g — the cost of garbage-collecting the cell? The crossover point is:

$$f = g$$

$$f = \frac{(c_1 + c_2 s)A}{M/s - A}$$

$$f = \frac{c_1 + c_2 s}{M/sA - 1}$$

$$M/sA = (c_1 + c_2 s)/f + 1$$

---

* Though it is technically possible to pop many cells (stack frames) from a stack at once by a subtraction from the stack pointer, this is usually very difficult to do in practice: it would entail returning from many procedures at the same time.

If M/sA is made larger than this, then garbage collection becomes cheaper than explicit freeing.

As an illustration, assume values for the constants c_i, f, and s, in some imaginary implementation:

$$c_1 = 3 \text{ instructions}$$
$$c_2 = 6 \text{ instructions}$$
$$s = 3 \text{ words}$$
$$f = 2 \text{ instructions}$$

Then we find that the crossover point is at

$$M/sA = 7$$

This is the ratio of physical memory to (average) reachable data. If one is willing to use 7 times as much memory as one has data, then garbage collection becomes essentially free.

Actually, M is really one-half of the machine's memory, since it is the size of each of the two spaces. Some enhancements of the copying algorithm use N spaces, with only one of them unused at a time, so there need not be this additional factor of two[4]. Furthermore, if the size of the data is truly much less than the size of each space, then the spaces could be made to overlap: the most memory needed at any one time is really $M + sA$, not $2M$.

Given a program that uses 2 megabytes of real data, all we need to do is run it in about 16 megabytes of physical memory to achieve ''free'' garbage collection.

4. The Massive Memory Machine

The Massive Memory Machine project at Princeton is designed to allow experiments to test performance improvements as memory increases. The current machine is a 128-megabyte VAX-11/785.

To confirm the results predicted in the previous section — that garbage collection time goes down dramatically as memory increases — a simple experiment was run: the Edinburgh Standard-ML[5] system compiling 2277 lines of Standard-ML code. This system is designed for portability rather than optimality, so it's not particularly fast; its garbage collector is written in C, not in assembly language; but even so, it provides a convincing demonstration of the advantages of massive memory.

Four runs were done of the same compilation, with different memory sizes: 4, 8, 16, and 64 megabytes.

Memory	M	CPU time	GC time	# GC's	time/GC	$M/sA - 1$ (typical)
4 Meg	2 Meg	663 sec	232 sec	34	7 sec	0.5
8	4	467	36	7	5	2.5
16	8	439	8	3	3	6.7
64	32	431	0	0	*	29

Because the two-space memory organization is used, the 4 megabyte run actually used only 2 megabytes at a time, and so on. The CPU time is the user time of the process; all of the runs had similar system times. The GC time is calculated just by subtracting the time of the 64-megabyte run from the times of the others (since the last run never garbage-collected); this may not be completely accurate because of caching effects, but it should be close.
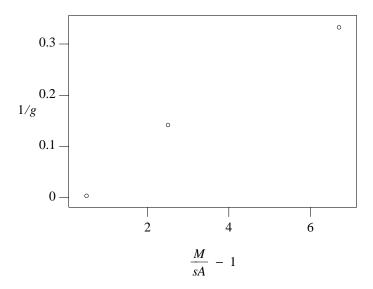
The number of garbage collections declines as expected; and more importantly, the average time per garbage collection does not increase as M does; it actually decreases. This is probably explained by the fact that in the first run, at times when sA approached M, the garbage collector started ''thrashing,'' repeatedly collecting when there was little garbage to collect.

Note that the last column, $M/sA - 1$, which measures the ratio of excess memory to reachable data, is not strictly linear in M. This is because the measurement is that taken at garbage collection time; and the garbage collections occurred at different points in the different runs.

It might seem that the table supports the conclusion that with enough memory, the garbage collector will never be invoked, and thus there is a time savings. One may consider the 64-megabyte run as useful

only to calibrate the measurement, however, and come to the much stronger conclusion: As memory increases, the number of garbage collections becomes proportionately fewer, and the time per garbage collection does not increase.

Section 2 predicts that garbage collection time should be inversely proportional to $M/sA - 1$. The graph of $1/g$ versus $M/sA - 1$ supports the conclusion that total garbage collection time is approximately inversely proportional to excess memory:

5. Allocating from the heap

The previous section shows that freeing a heap cell by garbage collection is cheaper (if there is enough memory) than popping a stack. Now consider the cost of allocating a cell. In LISP, this is typically expressed as **(cons A B),** meaning ''allocate a two-word cell 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 not a 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) to the free-space pointer.
   4. Store A into new cell.
   5. Store B into new cell.
   6. Return current value of free-space pointer.

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

This instruction sequence seems much more costly than the corresponding instructions for pushing **A** and **B** onto a stack. But 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 the 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   A,-(fp)
2.  movl   B,-(fp)
```

At this point, the pointer to the new cell is available in the free-space pointer (fp) 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.

This two instruction sequence to implement **cons** is identical to the sequence that pushes the values **A** and **B.** Thus, allocating a cell from the heap is identical in cost to pushing a cell onto the stack. With appropriate modifications to this simple idea, procedure-call frames may be heap-allocated just as cheaply as they can be allocated on a stack. And, as shown in the previous section, the overhead for deallocation of the cell can be made to approach zero; popping from a stack takes at least one instruction.

6. Remarks and Conclusions

A programming language with garbage collection allows a simpler and cleaner programming style than one in which the programmer must always return cells to the heap explicitly. It is easy to believe that one must pay a price in efficiency for this ease in programming; that explicit freeing of cells, though painful, yields a faster program. But this is simply not true. Even with an old and simple garbage collection algorithm, the cost of garbage collection can be made negligible. LISP programmers who go out of their way to avoid **cons**ing are muddying their programs needlessly.

The algorithm analyzed in this paper is a stop-and-copy algorithm, not a concurrent one. Thus, it has no guaranteed upper bound on response time, which is a disadvantage in a real-time environment. However, the total garbage collection overhead of this algorithm can be much less than that of a concurrent algorithm like Baker's[6], which has several instructions of overhead per cell allocated.

There are other algorithms that also become cheaper in proportion to the amount of memory in the system; the non-concurrent version of the Lieberman-Hewitt algorithm[4] is an example. Any system that incorporates such a garbage collector will automatically improve in speed as memories become cheaper and larger. But in a time when 50 megabytes of memory chips can be obtained for under four thousand dollars[7], there is less need for complex garbage collection algorithms, or special garbage collection hardware. Such techniques as reference-counting[8], ephemeral garbage collection[9], closure analysis[2], etc., may not really be necessary now that it is possible to use massive memories.

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. Guy L. Steele, ''Rabbit: a compiler for Scheme,'' AI-TR-474, MIT, 1978.

3. 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.

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. Robin Milner, ''A proposal for Standard ML,'' *ACM Symposium on LISP and Functional Programming*, pp. 184-197, ACM, 1984.

6. H. G. Baker, ''List processing in real time on a serial computer,'' *Communications of the ACM*, vol. 21, no. 4, pp. 280-294, ACM, 1978.

7. ''Oki goes to court on Microtech pact,'' *Electronic News*, vol. 32, no. 1609, July 7, 1986.

8. G. E. Collins, ''A method for overlapping and erasure of lists,'' *Communications of the ACM*, vol. 3, no. 12, pp. 655-657, ACM, 1960.

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