Vectorized Garbage Collection

*Andrew W. Appel**
Aage Bendiksen†

CS-TR-169-88

July 1988
Revised September 1988

ABSTRACT

Garbage collection can be done in vector mode on supercomputers like the Cray-2 and the Cyber 205. Both copying collection and mark-and-sweep can be expressed as breadth-first searches in which the "queue" can be processed in parallel. We have designed a copying garbage collector whose inner loop works entirely in vector mode. We give performance measurements of the algorithm as implemented for Lisp CONS cells on the Cyber 205. Vector-mode garbage collection performs up to 9 times faster than scalar-mode collection — a worthwhile improvement.

* Supported in part by NSF Grant DCR-8603543 and by a Digital Equipment Corp. Faculty Incentive Grant.

† Supported by the NSF Research Experiences for Undergraduates program.

1. Breadth-first garbage collection algorithms

Most garbage collection algorithms operate by traversing the graph of reachable nodes. *Copying* algorithms move the reachable nodes to a new area of memory during this traversal, whereas *mark-and-sweep* algorithms leave the traversed nodes where they are and (in a separate pass) put all the untraversed nodes onto a free list.

Any traversal that reaches all the reachable nodes can be used. Two common algorithms for traversing graphs are *depth-first search* and *breadth-first search.* The former uses a stack (last-in first-out data structure) to store nodes that have been seen but not examined, and the latter uses a queue (first-in first out) for the same purpose. Either kind of search can be used for garbage collection.

Some computers have highly pipelined vector instructions, in which the same instruction is applied to a series of inputs. The advantage of the vector instructions is that they can compute N results much more quickly than a sequence of N scalar instructions. Their disadvantage is that they are more difficult to make use of, since there is limited opportunity for control flow or special-case handling in the midst of a vector instruction. Traditional vector machines have instructions like vector-add, which can perform N additions in series, etc. Some of the more modern machines (e.g. the Cray-2 or the Cyber-205) have *gather* and *scatter* instructions, which can do random-access fetches (and stores, respectively) in vector mode. It is these machines that have the ability to do garbage collection in vector mode.

Depth-first search cannot be easily vectorized[1]. However, processing the queue in a breadth-first search algorithm is a natural application of parallel processing. A large batch of queue entries can be removed at once from the head of the queue, processed in parallel, and a new batch of entries can be appended to the tail. This is the principal idea behind our algorithm.

Since copying collection is much more efficient (in large enough memories) than mark-and-sweep collection[2, 3], we present an algorithm for vectorized copying collection, though the same idea easily applies to mark-and-sweep collectors.

2. Cheney's algorithm

The ''standard'' breadth-first copying garbage collection algorithm is due to Cheney[4]. We use two equal-size regions of memory. The mutator (the program making use of dynamic memory allocation) allocates new records contiguously in region 1; when it is full, the garbage collector is invoked. The collector copies the live data from region 1 (the ''source region'') into region 2 (the ''destination region''); then the roles of the regions are swapped, and the mutator can allocate records from the rest of the region 2 until it fills up and the garbage collector is invoked again.

The copying can be done without any auxilary data structure by incorporating the breadth-first search queue into the destination region. 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 beginning of the region. For simplicity, assume that each record contains two fields, R[1] and R[2], each of which may be a pointer or a non-pointer.

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 R[1] to location NEXT
                      copy R[2] to location NEXT+1
                      assign NEXT into R[1]
                      increment NEXT by 2
                      return R[1]
          else return 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.

The ''queue'' is simply the area between *scan* and *next*; when the ''head'' (*scan*) catches up with the ''tail'' (*next*), the queue is empty. Note that the scanning procedure can ignore record boundaries in forwarding each of the pointers between *scan* and *next*.

3. Processing the queue in parallel chunks

To vectorize Cheney's algorithm, we can grab k elements from the head of the queue (at *scan*) and scan them in parallel. This will result in a batch of up to k records (or sk pointers, where s is the number of fields in a record) being added to the tail of the queue.

Here is the inner loop of the algorithm in detail, with each item corresponding roughly to one vector instruction on the Cray or Cyber:

1. Let the k words starting at *scan* be called *original.*
2. Determine *ptrs* and *nonptrs*, the elements of *original* that are pointers into source-space, and the non-pointer elements, respectively.
3. Gather, by the addresses in *ptrs*, into *first*. These are the first words of each cell referred to by the k words after *scan*. (The *gather* instruction is given a vector of addresses and fetches each one, producing a corresponding vector of data from memory.)
4. Determine *forwarded*, the elements of *first* that point into to-space, and *non-forwarded*, the other elements of *first*. Then determine *copy*, the elements of *ptrs* that correspond to *non-forwarded*. These are pointers to the cells in destination space that must be copied, as they don't contain forwarding pointers.
5. Make an iota vector with base of *next* and a stride of 2, called *new* (i.e., *new* is a vector of addresses $next, next+2, next+4, \cdots$). These are the addresses that the cells *copy* must be moved to.
6. Add 1 to each address in *copy*, and gather into *second*. This grabs the second word of each cell to be copied.
7. Store *first* starting at *next* with a stride of 2. Store *second* starting at $next+1$ with a stride of 2. This copies the cells from the source space to the destination space.
8. Scatter *new* by the addresses *copy*. This installs forwarding pointers in the source-space cells that have been copied. (The *scatter* instruction is the opposite of *gather*: given a vector of data and a

vector of addresses, each datum is stored into memory at the corresponding address.)

9. Then gather by the addresses *copy* into new_1. This is necessary in case there were several references to the same cell; in this case case the same address appears more than once in *copy*. There will be more than one copy of the cell after *next*. However, we require that all references to this cell point to the same copy. During step 8, any address that appears more than once in *copy* will have been written to more than once; but just one value will end up in the memory location. This will provide a unique address for the copied cell, and the gather will put that address into all the appropriate positions of new_1. The unused copies (after *next*) won't affect the correctness of the algorithm, as long as all the references to the copied cell point to the same copy.

10. Merge new_1 (pointers to cells just copied) with *forwarded* (pointers to cells copied in previous phases), and merge the result with *nonptrs* (words of *scan* that weren't pointers); write back into the *k* words after *scan*. (A *merge* instruction takes a vector of booleans, and a vector of data whose length is equal to the number of true elements of the boolean vector. The data is written to sequential addresses, except that wherever a false value appears in the boolean vector, an address is skipped.)

11 Increment *scan* by *k*, and *next* by twice the length of *copy*. This completes one iteration of the algorithm, which may be continued while *scan>next*.

This inner loop can be written in Vector-C[5], a language supported on the Cyber 205 supercomputer. In Vector C, the notation `a[0#s]` represents a vector *a* of length *s*, indexed by 0 through $s-1$. The expression `a[b[0#r]]` as an r-value represents a gather, and as an l-value represents a scatter, assuming that *b* is a vector of integer indexes. If *b* is a vector of booleans, then `a[b[0#r]]` as an r-value is a *compress*, selecting only those elements of *a* corresponding to *true* elements of *b*; as an l-value it is an *expand*, reversing this operation. The `@||` operator counts the *true* elements of a vector of booleans.

The line numbers in this Vector-C program correspond to the steps in the description above:

```
/* Forward copies in destination space */
    while (scan < next) {
 1.     K = ((next - scan) > maxK) ? maxK : next - scan;
 2.     is_ptr[0#K] = (scan[0#K] >= (int) source) && (scan[0#K] < srctop);
        L = @|| is_ptr[0#K];
        if (L > 0) {
         ptrs[0#L] = scan[is_ptr[0#K]]/64;
 3.      first[0#L] = virt_addr[ptrs[0#L]];
 4.      is_frwed[0#L] = (first[0#L] >= (int) dest) && (first[0#L] < dsttop);
10.      if (is_frwed[0#L]) write[0#L] = first[0#L];
 4.      non_frwed[0#L] = (first[0#L] < (int) dest) || (first[0#L] >= dsttop);
         L2 = @|| non_frwed[0#L];
         if (L2 > 0) {
          non_forwarded[0#L2] = first[non_frwed[0#L]];
          copy[0#L2] = ptrs[non_frwed[0#L]];
 5.       new[0#L2] = iota[0#L2] + (int) next;
 6.       secondnf[0#L2] = virt_addr[copy[0#L2] + 1];
 7.       next[0#L2:2] = non_forwarded[0#L2];
          next[1#L2:2] = secondnf[0#L2];
 8.       virt_addr[copy[0#L2]] = new[0#L2];
 9.       newnf[0#L2] = virt_addr[copy[0#L2]];
10.       write[non_frwed[0#L]] = newnf[0#L2];
11.       next += RECSIZE * L2;
         }
10.       scan[is_ptr[0#K]] = write[0#L];
        }
11.     scan += K;
    }
```

4. Benchmarks and analysis

We implemented both a scalar-mode garbage collector and a vector-mode collector on the Cyber 205. We have no programming environment on that machine that requires a garbage collector, but that is not necessary to get useful measurements of performance; Cheney's algorithm takes time proportional to the number of copied cells, and we just wish to measure the constant of proportionality.

We ran the collectors on two different inputs: one was a large Fibonacci tree structure with no sharing of nodes, the other was a set of 128 linear lists.

Algorithm	Input	Vector Length	Words copied	CPU time	Million Words/Second
Scalar	Tree	-	408,576	1.07 sec	0.382
Scalar	Lists	-	255,488	0.683	0.374
Vector	Tree	16	408,576	1.151	0.355
Vector	Lists	16	255,488	0.728	0.351
Vector	Tree	64	408,576	0.363	1.127
Vector	Lists	64	255,488	0.219	1.167
Vector	Tree	2048	408,576	0.114	3.584
Vector	Lists	2048	255,488	0.102	2.504

Table 1.

Table 1 shows the benchmark data. When the maximum vector length, was limited to 16, the vector algorithm gave performance comparable to the scalar algorithm. With longer vectors, however, the vector algorithm ran up to 9 times faster than the scalar algorithm. This is a very significant speedup indeed.

The last two lines of the table show the performance of the vector algorithm with (practically) unlimited vector length. On the ''tree'' input the algorithm performs significantly faster than on the ''lists'' input. This is undoubtedly because the lists are ''narrower'' than the tree; the queue of the breadth-first search (the difference between *scan* and *next*) never grows to more than 256, limiting the effective vector size to 256. This limits the performance of the vector algorithm, though it still outperforms the scalar algorithm by an order of magnitude.

Another quantity of interest is the number of ''wasted'' words; those allocated by step 5 of the algorithm but then discarded in step 9. Unfortunately, this seems to be very input-dependent, and it's hard to provide realistic estimates of it without running a real programming environment. We constructed an input similar to our ''tree'' but with one fifth of the pointers sharing common subexpressions; we ran the algorithm on this input with different vector lengths, counting the number of wasted words (Table 2).

Vector Length	Words Copied	Wasted Words	Wasted %
1	215,602	0	0%
16	217,960	2,358	1.1%
32	218,282	2,680	1.2%
64	218,448	2,846	1.3%
128	218,520	2,918	1.3%
65535	218,604	3,002	1.4%

Table 2.

Clearly, the wasted cells pose no great concern.

5. Variable-sized records

The algorithm is easy to describe and easy to implement for fixed-size records. It is not too difficult to adapt vectorized collection to variable-sized records.

Let us examine the distribution of record sizes in a system with records of varying size. Table 3 shows the distribution of record sizes observed as the Standard ML of New Jersey compiler[6] compiles itself. A total of 30838749 records were created, most of size two or three; in addition, 4675 arrays of average size 74 were created.

Size of Record	Frequency	Cumulative total
1	2.8%	2.8%
2	74.1	76.9
3	12.4	89.3
4	1.5	90.9
5	1.3	92.2
6	3.6	95.9
7	1.0	96.9
8	0.0	96.9
9	0.1	97.0
10	0.3	97.4
11	0.2	97.7
12	2.0	99.7
>12	0.2	100.0

Table 3.

Almost all records are less than 13 words long. This suggests that a collector could handle records of up to 12 words in vector mode, use use special-case code for longer records, and still achieve high performance.

Let S maximum size of commonly-occurring records (i.e. $S=12$). By using vector compression, the collector can quickly sort the vector *ptrs* into S different subvectors, and handle each subvector as the original algorithm handles the *ptrs* vector.

What remains is a short subvector of large, odd-sized records. A loop can traverse this subvector; each of the large records can be copied using sequential vector-reads and vector-writes.

6. Conclusion and remarks

For fixed-size records, vector-mode garbage collection performs very well, providing an order-of-magnitude speedup over scalar-mode collection. The algorithm is adaptable to environments with records of varying size, and should still yield high performance.

This algorithm is compatible with generational garbage collection schemes[2] in which many fewer cells are copied because the effort is concentrated on the most volatile areas.

References

1. John H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, pp. 229-234, 1985.
2. 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.
3. A. W. Appel, "Garbage collection can be faster than stack allocation," *Information Processing Letters*, vol. 25, no. 4, pp. 275-279, 1987.
4. C. J. Cheney, "A nonrecursive list compacting algorithm," *Comm. ACM*, vol. 13, no. 11, pp. 677-678, 1970.
5. "CDC Cyber 200 Vector C," Publication #60000018, Control Data Corp., 1986.
6. 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.