Fast garbage compaction with interior pointers

Benedikt Meurer
Compilerbau und Softwareanalyse
Universität Siegen
D-57068 Siegen, Germany
meurer@informatik.uni-siegen.de

Abstract

This paper presents a garbage compaction algorithm, which extends the well-known algorithm of Jonkers with the ability to handle interior pointers. It is as efficient as Jonkers' algorithm in the absence of interior pointers, and in practice only slightly less efficient in the presence of interior pointers, while at the same time, it does not impose any additional space overhead. This, however, is partly due to the fact that only certain kinds of interior pointers are allowed.

1 Introduction

Compacting garbage collection is a useful technique to implement heap storage management in modern runtime environments. The need for compaction of heap storage and a growing number of compaction techniques have been presented previously.

Today there are basically two classes of compaction algorithms. The first class employs a new area as large as the original heap area and compact the heap by moving live block to a contiguous segment of the new area. Algorithms in this class are usually referred to as *copying garbage collectors* or *semi-space garbage collectors* [2, 3, 7, 14, 16].

The other class generally traces and marks all storage in use, plan where the live blocks of storage are to be moved, update each pointer to point to the planned address of the referenced block, and finally move each block to its planned address. Algorithms in this class are usually referred to as *mark-compact garbage collectors* [8, 9, 10, 12, 13, 15, 17, 18].

This paper deals with the well-known compaction algorithm presented by Jonkers [12], which compacts live blocks within a single contiguous heap space by sliding them to the low end of the heap space. This algorithm is one of the most popular garbage compactors because it requires only two heap passes, does not impose any additional space overhead and is relatively easy to adapt. However – in its basic form – it cannot cope with pointers to the interior of live blocks.

We will present an extension to the basic algorithm, which adds support for pointers containing addresses of special interior cells, called *interior header cells*. We start by defining the problem in section 2. Section 3 describes the basic algorithm. Then, in section 4, we present our extended algorithm. Section 5 reviews the efficiency of the extended algorithm. Finally section 6 compares our work with related compaction algorithms.

2 Problem

Following the original problem statement in [12] we represent a machine memory by an array M of *cells*. Every cell has a unique *address*, which is the index of the cell in M. A *pointer* is an address or the special value **nil**, which is not equal to any address in M. We assume that each cell is large enough to contain a pointer.

There is a subarray S of M with indices $s_1, \ldots, s_k$, called the *store*, which is the part of M to be compacted. The store S contains a number of disjoint subarrays called *nodes*. Every node contains exactly one *header cell*, zero or more *interior header cells* and a number of *pointer cells*, which are cells that contain a pointer. The *header address* of a node is the address of its header cell. An *interior address* of a node is the address of one of

its interior header cells. Every node thereby has one or more *addresses*, formed by the *header address* and the *interior addresses*. For the sake of simplicity we take the first cell of a node to be the header cell.

We assume that the contents of the header and interior header cells are distinguishable from a pointer. Furthermore we assume that a pointer q in a pointer cell of a node is either **nil** or points to a node, i.e. q is either **nil** or an address of a node.

Just before the call to the compaction routine, the marking phase of the garbage collector will have established the following information:

1. The addresses $r_1, \ldots, r_n$ of cells not contained in S, which contain either **nil** or a pointer to a node. These addresses are called *root set* and form the starting points of the marking phase.

2. The header addresses $l_1, \ldots, l_m$ of the nodes contained in S. These nodes have been determined by the marking phase as being *live* (i.e. non-garbage).

This situation is schematically shown in Fig. 1.

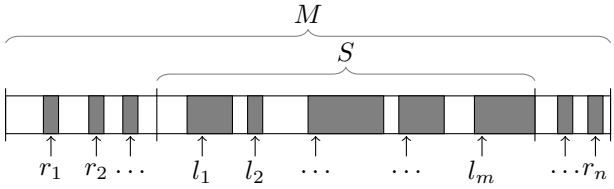

Figure 1: Machine memory after marking phase

The problem now is to rearrange the nodes in S in such a way that

1. the nodes occupy a contiguous memory area at the lower end of S;

2. (interior) pointers in the root set $r_1, \ldots, r_n$ and pointers in the pointer cells of nodes in S still point to the same (interior) headers of the same nodes as before;

3. the order of nodes in S is preserved.

3 Jonkers' algorithm

In this section we describe the original algorithm proposed by Jonkers [12]. It is based on a technique called *threading* first proposed by Fisher in [8], which is also used in [5, 11, 15, 17].

Jonkers' algorithm assumes that no interior header cells are present. That is, all pointers within the root set and the pointer cells of the nodes in S are either **nil** or point to the first cell of a node in S. More formally, the set of all such non-**nil** pointers is a subset of $\{l_1, \ldots, l_n\}$.[1]

We first describe the algorithm informally and illustrate it by the example given in [12]. The initial situation is given in Fig. 2(a), where only the pointers to an single node q are shown.

The algorithm starts by visiting the cells in the root set $r_1, \ldots, r_n$. For each root cell r containing a pointer to a node q, r cannot be updated immediately since the new address of the node is not yet known. Therefore r is *threaded* to q. For now it does not matter how this threading is achieved. We will discuss that later.

Fig. 2(b) shows the situation once all cells in the root set have been visisted this way, where the dashed arrow indicates the *is threaded to* relation. The algorithm continues by scanning the store S twice, visiting all pointer cells of all nodes in the order from lower addresses to higher addresses. Upon visiting a node q in the first scan (Fig. 2(c)), the address where q is to be moved to is known from an accumulated counter. Using this information all pointer cells threaded to q at this point are updated as shown in Fig. 2(d), where *updating* a cell also means *unthreading* it.

Subsequently all pointer cells of q containing a pointer to a node are threaded to that node (Fig. 2(e)). Once the first scan is done all pointer cells in the root set will have been updated this way. Furthermore, all pointer cells pointing *forward* to q – in ascending address order – will have been updated as well, and all pointer cells pointing *backward* to q, including pointer cells of q itself, will be threaded to q (Fig. 2(f)).

The second scan visits all nodes in ascending address order again. Upon visiting a node q (Fig. 2(g)), all pointer cells threaded to q must be backward pointers, and neither q nor any pointer cell threaded to q will have been moved yet. Therefore all cells threaded to q can be correctly updated to the new address of q as shown in Fig. 2(h). Once

[1] These sets will be equal if the marking algorithm is accurate.

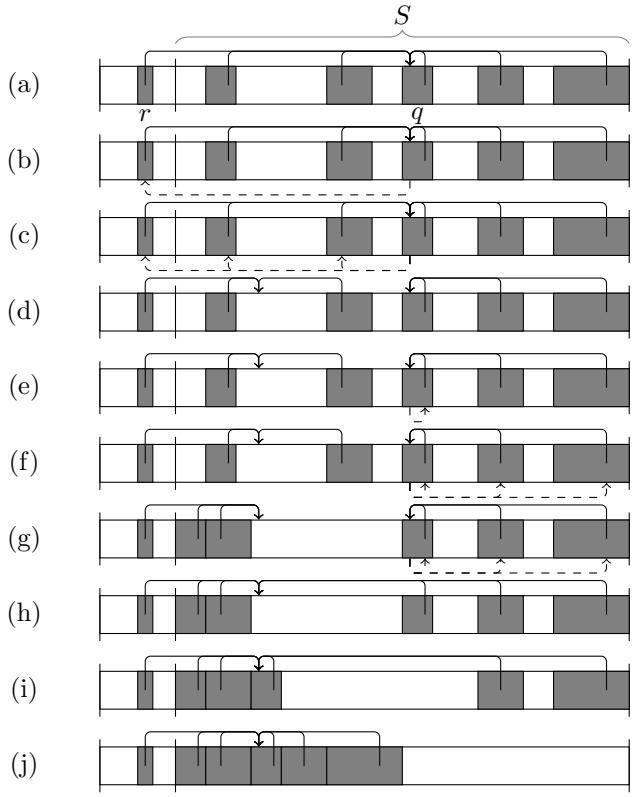

Figure 2: Example run of Jonkers' algorithm

this is done, all cells pointing to the initial address of q will have been updated to the new address of q, and all nodes to the left of q will have been moved to their new locations already. Therefore it is now safe to move q to its new location (Fig. 2(i)) as well. Once the second scan is complete, all nodes will have been moved and all pointers will have been updated (Fig. 2(j)).

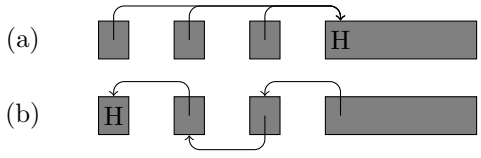

Figure 3: Example of threading

In order to fully understand the algorithm, we have to explain how *threading* works in detail, especially why this threading of cells can be done without any space overhead using the well-known trick described in [8]. By assumption and from the (informal) description above we know two things:

1. The first cell of a node contains the header, which is distinguishable from any address.

2. All pointer cells threaded to a node initially contain a pointer to that node.

Now consider the situation shown in Fig. 3(a) with 3 cells holding pointers to a node. We can transform this situation in such a way that all cells threaded to a node are chained in a list, using the header cell of the node as list head and the original content of the header cell as list terminator (Fig. 3(b)). This transformation can be performed without loss of information, since both the original content of the header cell as well as all information about the cells pointing to it is preserved within the structure of the list.

Updating all cells threaded to a node is now a matter of traversing this list, updating the cells in it and restoring the header value of the node using the

list terminator. It is especially important to restore the original header value while scanning a node, since it may be necessary to determine the size of the node as well as its pointer cells (during the first scan) or to move the node (during the second scan).

Algorithm 1 Jonkers' garbage compactor

```
procedure THREAD(p)
    if M[p] ≠ nil then
        q ← M[p]; M[p] ← M[q]; M[q] ← p
    end if
end procedure

procedure SCAN(p)
    let p_1, ..., p_n be the addresses of the
    pointer cells of the node with address p;
    for i ← 1, ..., n do
        THREAD(p_i)
    end for
end procedure

procedure UPDATE(old, new)
    p ← M[old]
    while ISADDRESS(p) do
        q ← M[p]; M[p] ← new; p ← q
    end while
    M[old] ← p
end procedure

procedure MOVE(old, new)
    for i = 0, ..., SIZE(old) − 1 do
        M[new + i] ← M[old + i]
    end for
end procedure

procedure COMPACT()
    for i ← 1, ..., n do
        THREAD(r_i)
    end for
    new ← s_1
    for i ← 1, ..., m do
        UPDATE(l_i, new); SCAN(l_i)
        new ← new + SIZE(l_i)
    end for
    new ← s_1
    for i ← 1, ..., m do
        UPDATE(l_i, new); MOVE(l_i, new)
        new ← new + SIZE(l_i)
    end for
end procedure
```

The whole procedure is described formally in Algorithm 1 using pseudocode.

4 Our algorithm

Jonkers' original algorithm – as described in the previous section – does not work if pointers are allowed to point not only to header addresses but also to interior header addresses. This is because interior headers are ignored by the UPDATE procedure. If there would be a way to reliably locate all interior headers of a node, fixing this shortcoming would be a matter of adjusting the UPDATE procedure to also update pointer cells threaded to the interior headers. However it may not be feasible, indeed it may even be impossible, to implement this in practice.

Instead, we assume that, given a pointer to an interior header, it is possible to determine the node, and hence the header of the node, to which this interior address belongs. This is a realistic assumption and might also be necessary for the marking algorithm anyway.

In this section we will present an extension to Jonkers' algorithm, which is able to thread pointers to interior headers of a node and update them properly to the new address of the interior header to which the node will be moved. Our extension shares the good properties of the original algorithm, which means it does not impose any additional space overhead and requires linear time in practice. It does however place additional restrictions on the shape of pointers and cells. Furthermore it assumes that, given the node header and the interior offset, it is possible to reconstruct an interior header.

Consider the situation given in Fig. 4(a) with three pointers r_1, r_2 and r_3 pointing to a node, where r_1 points to the header H of the node while r_2 and r_3 point to the interior headers IH_1 and IH_2 respectively. Upon visiting r_1 it can be threaded to the node as described in the previous section (Fig. 4(b)).

However the pointer cell r_2 cannot simply be threaded to the interior header IH_1, since by assumption we are unable to locate all interior headers of a node, and as such the pointer cells threaded to an interior header of a node would not be updated.

The basic idea of our algorithm is to thread interior pointer cells to the node header just like cells pointing to the header of a node. Since the interior pointer cells need special treatment during update we chain both the interior header cell and the interior pointer cell, while tagging the pointer to the interior header in special way as shown in Fig. 4(c). Once all pointers and interior pointers to a node are threaded, the situation looks as given in Fig. 4(d).

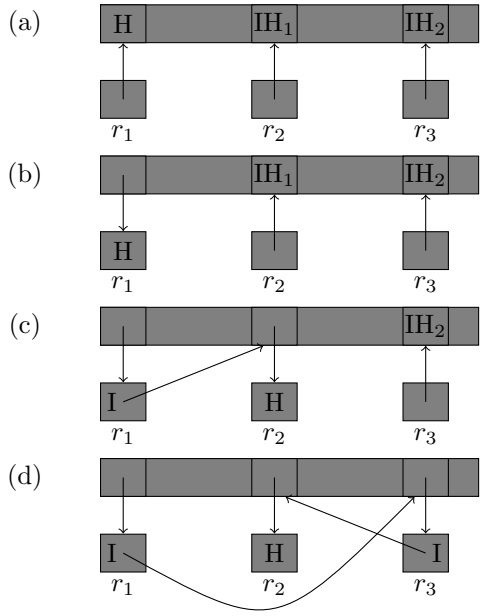

Figure 4: Example of threading with interior pointers

When updating cells we replace pointers with the *new* address of the node as described in Jonkers' algorithm. Upon reaching a tagged pointer we know that the cell pointed to originally contained an interior header and the remaining cells in the chain (up to the terminating header or up to the next tagged pointer) originally pointed to this interior header. Therefore we reconstruct the interior header, store it into the cell pointed to by the tagged pointer, set *new* to the address of the interior header at the new location of the node and continue updating the remaining cells in the list.

While the original algorithm requires the ability to distinguish header values from addresses, our algorithm imposes additional constraints. This is because threading now requires the ability to distinguish headers and interior headers, and updating now requires the ability to distinguish headers, threaded pointers and pointers to interior header cells. To sum up, we need to distinguish the following:

1. Values of node headers
2. Pointers to node headers
3. Values of interior headers

Algorithm 2 Threading with interior pointers

```
procedure THREAD(p)
    q ← M[p]
    if ISINTERIORHEADER(M[q]) then
        o ← NODEOFINTERIORADDRESS(q)
        while ISPOINTER(M[o]) do
            o ← M[o]
        end while
        M[p] ← M[o]
        M[q] ← p
        M[o] ← INTERIORPOINTER(q)
    else
        M[p] ← M[q]
        M[q] ← p
    end if
end procedure

procedure UPDATE(old, new)
    h ← M[old]
    while not ISHEADER(h) do
        h ← M[POINTER(h)]
    end while
    p ← M[old]
    n ← new
    while not ISHEADER(p) do
        if ISPOINTER(p) then
            q ← M[p]
            M[p] ← n
        else                          ▷ p must be an interior pointer
            p ← POINTER(p)
            n ← new + (p − old)
            q ← M[p]
            M[p] ← MAKEINTERIORHEADER(h, old, p)
        end if
        p ← q
    end while
    M[old] ← p
end procedure
```

4. Pointers to interior headers

Given that we are able to distinguish the above, we can implement the changes necessary to handle interior pointers as shown in Algorithm 2. The rest of the algorithm remains unchanged.

It is easy to see that the new algorithm is correct. Cells with pointers to the node header will be chained to the beginning of the threaded list before any cells with pointers to interior headers. So these pointers will be updated with the *new* location of the node.

Cells with pointers to interior headers will always be linked in between the tagged pointer that contains the address of this interior header and the next tagged pointer that contains the address of another interior header (or the terminating node header). Furthermore since interior headers

threaded to a node are replaced with pointers, the same interior cell cannot be threaded more than once. Therefore each interior header cell on the list is updated exactly once and the pointer cells threaded to it will be updated to the correct interior address at the new location of the node.

5 Efficiency

The original algorithm visits every node twice, and performs at most one test, thread and update operation per pointer cell, and exactly one move per node. This results in a worst-case time complexity of $O(k + n)$, where k is the number of cells in the store and n is the number of cells in the root set. Since the root set is usually smaller than the store, Jonkers' algorithm is said to be linear in the size of the store.

The extended algorithm performs the same operations. However, in contrast to the original algorithm, THREAD has to traverse the list of previously threaded pointer cells once for each interior header being pointed to. The worst-case time complexity is therefore $O(k \cdot (k + n))$. This seems to be a major regression compared to the original algorithm upon first sight. However, in practice the number of interior headers is probably low[2]. Furthermore it has been observed that most live objects in modern applications have only a single referent [1, 4, 6, 19]. Hence, within a realistic application, the THREAD procedure requires constant time, and as such, the whole algorithm is linear in the size of the store.

Moreover, under the assumption that appropriate header and interior header cells are available for the cells, the extended algorithm requires no space overhead. Since each node usually has to carry additional information such as size and type of the node content, the runtime environment will have to reserve space for this kind of information anyway (usually as part of a header word). Interior headers on the other hand might require some extra effort, and will usually carry information such as the offset of the interior address relative to the address of the first cell. This kind of information is often already required by the marking phase of the garbage collector.

[2]Note that the worst-case time complexity of the extended algorithm is still $O(k + n)$ in the absence of interior headers.

The algorithm as described here leaves some room for performance improvements with respect to effective garbage collection time. For example, the UPDATE routine as shown in Algorithm 2 first traverses the list of threaded cells to determine the original header of the node. This information is used to reconstruct the interior headers within this block. However, since interior headers should be rare, it might be better to determine this information only on-demand. In fact, it might even be possible to get by without this additional list traversal if the reconstruction algorithm does not need the original node header at all (i.e. if the interior header contains only the offset of the interior cell relative to the first cell of the node).

The additional requirement to distinguish four different kinds of values is also rather simple to fullfil in practice. Under the assumption that all words are aligned on 4-byte address boundaries (which may already be required by the addressing constraints of the target machine), one can use the two least significant bits of machine words to distinguish 4 different kinds of values.

6 Related work

There exists a different class of moving compaction algorithms, which perform pointer adjustments based on range checks within a *break table* [9, 10, 13, 18], and therefore support interior pointers *out of the box*. However these algorithms either do not operate in time linear to the size of the store [9, 10, 18], or require substantial space overhead [13, 18].

The algorithm presented in [15] is somewhat similar to the basic algorithm described in section 3, and is capable of updating arbitrary interior pointers. But it is less efficient than the algorithm described here and requires additional space overhead (one mark bit per cell).

7 Conclusions

We have described an algorithm that extends Jonkers' fast garbage compaction algorithm with the ability to handle pointers to certain interior cells of live objects without any additional space overhead. The extended algorithm is linear in the

size of the store in practice and requires one additional bit per value – compared with the requirements of the original algorithm – to differentiate four kinds of cell values relevant for threading and updating the pointer cells.

Acknowledgements

We would like to thank Kurt Sieber and Christian Uhrhan for their careful proof-reading.

References

[1] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[2] C. J. Cheney. A non-recursive list compacting algorithm. *CACM*, 13(11):677–8, Nov 1970.

[3] D. W. Clark. An efficient list moving algorithm using constant workspace. *CACM*, 19(6):352–354, Jun 1976.

[4] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, 1976.

[5] R. B. K. Dewar and A. P. McCann. MACRO SPITBOL — a SNOBOL4 compiler. *SPE*, 7(1):95–113, 1977.

[6] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the specjvm98 java benchmark. In *ECOOP*, pages 92–115, 1999.

[7] R. R. Fenichel and J. C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *CACM*, 12(11):611–612, Nov 1969.

[8] D. A. Fisher. Bounded workspace garbage collection in an address-order preserving list processing environment. *Inf. Process. Lett.*, 3(1):29–32, 1974.

[9] J. P. Fitch and A. C. Norman. A note on compacting garbage collection. *CompJ*, 21(1):31–34, Feb 1978.

[10] B. K. Haddon and W. M. Waite. A compaction procedure for variable length storage elements. *CompJ*, 10:162–165, Aug 1967.

[11] D. R. Hanson. Storage management for an implementation of Snobol 4. *SPE*, 7(2):179–192, 1977.

[12] H. B. M. Jonkers. A fast garbage compaction algorithm. *Inf. Process. Lett.*, 9(1):26–30, 1979.

[13] B. Lang and B. Wegbreit. Fast compactification. Technical Report 25–72, Harvard University, Cambridge, MA, Nov 1972.

[14] M. L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, Dec 1963.

[15] F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Commun. ACM*, 21(8):662–665, 1978.

[16] E. M. Reingold. A non-recursive list moving algorithm. *CACM*, 16(5):305–307, May 1973.

[17] L.-E. Thorelli. A fast compactifying garbage collector. *BIT*, 16(4):426–441, 1976.

[18] B. Wegbreit. A generalised compactifying garbage collector. *CompJ*, 15(3):204–208, Aug 1972.

[19] M. Wegiel and C. Krintz. The single-referent collector: Optimizing compaction for the common case. *ACM Trans. Archit. Code Optim.*, 6(4):1–26, 2009.