FAST COMPACTIFICATION

by

Bernard Lang
Ben Wegbreit

November 1972

Center for Research in Computing Technology

Harvard University
Cambridge, Massachusetts 02138
FAST COMPACTIFICATION

by

Bernard Lang
Ben Wegbreit

November 1972
ABSTRACT

A technique is presented for compacting free storage by partitioning the area into regions. This allows fast updating of pointers, efficient re-use of storage, and adaptability to various address size/word size ratios.

Key Words and Phrases:
compactifying garbage collection, compaction, compactification,
free storage management, heap management, list processing,
storage allocation

CR Categories: 4.22, 4.49
1. INTRODUCTION

Compactifying garbage collection is useful in free storage management. The need for compactification and a number of compactification techniques have been presented previously.

These algorithms generally trace and mark all storage in use, plan where the blocks of storage are to be moved, update each pointer to point to the planned address of the referenced object, and move each block to its planned address. Determining for each pointer the planned address of the referenced object is the key step. For this, a map of relocation pairs (old address, correction term) is constructed. It is desirable to

1. store the map in unused words of the free storage area (since it is unpredictably large)
2. structure the map so the correct pair for a pointer can be found very rapidly
3. construct the map rapidly.

This paper shows how these objectives can be simultaneously realized (and under a variety of implementation constraints). Notation and assumptions are as in [10] unless stated otherwise.

2. THE ALGORITHM

The algorithm starts by tracing all active pointers, marking all words in use (e.g., as in [3,9]). Next, a linear sweep finds maximal contiguous blocks of free words, and records in each block its size and the address of the next free block, forming the free list (as in [6,10]). The left part of Figure 1 shows the result after this sweep. We assume there is a region of memory available to

---

* A second class of compactification algorithms employ a new area as large as the original free storage area and compactify by moving blocks in use to a contiguous segment of the new area. This paper treats compactification where such a new area is not available.

** The essential ideas of this paper apply to a variety of machines and allow a number of time/space trade-offs. Hence, Section 2 presents the algorithm abstractly, making assumptions about available resources. Section 3 discusses how these assumptions may be realized in a number of concrete situations. Assumptions made in Section 2 and resolved in Section 3 are marked with a 1 on their first appearance.
to be used as a free buffer.

Compactification is performed in four stages:

(1) Initialization

1.1) The heap (i.e., free storage area) is segmented into equal sized regions.

1.2) Space is reserved in the free buffer for a region directory and a temporary site for the relocation map.

(2) Regional Compactification

For each region in turn, the following is done.

2.1) The region is compactified by moving all used blocks toward low addresses to form a packed segment, leaving all free space at the high address end of the region.

2.2) Whenever a block is moved, a relocation pair for that block is added to the relocation map for the region, temporarily stored in the free buffer.

2.3) When all blocks have been moved, the relocation map is moved to the free space at the end of the region.

2.4) A region descriptor is added to the region directory.

(3) Updating Pointers

Each active pointer is adjusted to point to the final address after compactification of the referenced object.

(4) Global Compactification

The packed segments of all regions are moved to form a single contiguous block comprising all storage in use.

The next four subsections treat these stages in detail.

2.1 Definitions and Notation

Objects are considered at three stages during the algorithm: (1) after the initial trace and just before regional compactification, objects are at their original addresses, (2) between regional and global compactification – at interim addresses, (3) after final compactification – at final addresses.
Roman capital letters denote addresses while script capitals denote blocks of memory; Roman letters are unprimed, primed, or doubly primed as an address is original, interim, or final.

Figure 1 shows a heap at original, interim, and final stages. Numbers are in octal. For simplicity, a very small heap (60_8 words) and region size (10_8) is used.

Let \( \mathcal{U}_j \) denote the \( j \)th used block and let \( U_j \) and \( V_j \) be the original addresses of the first and last word of \( \mathcal{U}_j \). Let \( U'_j \) and \( V'_j \) (\( U''_j \) and \( V''_j \)) be the interim (final) addresses. The difference \( u'_j = U'_j - U_j \) is the interim relocation term; \( u''_j = U''_j - U_j \) is the final relocation term.

Let \( \mathcal{R}_i \) denote the \( i \)th region and let \( A_i \) and \( Z_i \) be the first and last word of the region. The block \( \mathcal{U}_j \) is said to belong to \( \mathcal{R}_i \) whenever \( A_i \leq U_j \leq Z_i \). Note that a region can be occupied partially or totally by a block belonging to a preceding region. A region to which no block belongs is said to be trivial.

Regional compactification of \( \mathcal{R}_i \) moves together all used blocks belonging to \( \mathcal{R}_i \) to form a packed segment which we denote by \( S_i \). \( S_i \) is immediately followed by the region relocation map denoted by \( M_i \). Let \( A'_i \), \( M'_i \) and \( Z'_i \) be the addresses of the first word of \( S_i \) and the first and last word of \( M_i \). The size of the relocation map is denoted by \( s_i = Z'_i - M'_i + 1 \). Let \( A''_i \) be the address of the first word of \( S_i \) in its final location. The difference \( r_i = A''_i - A'_i \) is the region relocation term. If block \( \mathcal{U}_j \) belongs to \( \mathcal{R}_i \), then \( u''_j = u'_j + r_i \), i.e., the final relocation term is the interim relocation term (regional compactification) plus the region relocation term (global compactification).

For each \( \mathcal{R}_i \), the relocation map \( M_i \) is an array of relocation pairs, one pair \( P_j \) for each block \( \mathcal{U}_j \) belonging to the region. \( P_j = (\hat{U}_j, u''_j) \) where \( \hat{U}_j \) denotes the address of \( \mathcal{U}_j \) relative to the beginning of the region, i.e., \( \hat{U}_j = U_j - A_i \). Assume that a relocation pair can be stored in one word. Hence, the size of \( M_i \), \( s_i \), is equal to the number of used blocks belonging to \( \mathcal{R}_i \). The relocation maps are built so that the pairs are ordered on their first component.
Figure 1
Thus, given an original address pointing into a block \( U_j \) belonging to \( R_i \), the corresponding relocation pair can be found by a binary search of \( M_i \).

The region directory \( D \) is an array of region descriptors. For each \( R_i \), the region descriptor \( D_i = (M'_i, s_i) \), i.e., the address of \( M_i \) and its size. Usually, a region descriptor can be stored in one word.

2.2 Regional Compactification

Regions are compactified in increasing order of their indices. Consider processing \( R_i \). Two cases arise, depending on whether or not \( R_i \) is trivial.

If \( R_i \) is nontrivial, the following steps are taken.

(1) The starting address \( A'_i \) of \( S_i \) is computed

\[
A'_1 = A_1 + 1^* \\
A'_i = \text{Max}(Z_{i-1}, Z'_{i-1}) + 1, \quad \forall i \geq 2
\]

\( A'_i \) is the address of the first word which does not belong to a preceding region and is not occupied by a used block or a map.

(2) The region relocation term \( r_i \) is computed as follows. Assuming ** the original and final low addresses of the heap are identical,

\[
r_i = 0
\]

For all \( i \geq 2 \), \( S_i \) is moved in global compactification by the relocation term of \( S_{i-1} \) plus the amount of free space to be eliminated between the segments, i.e.

\[
r_i = r_{i-1} + (M'_{i-1} - A'_i)
\]

(3) All used blocks belonging to \( R_i \) are moved toward lower addresses, packing them into one segment beginning at \( A'_i \). (It is shown in the Appendix that if \( U_{a_i} \) is the first block belonging to \( R_i \), \( A'_i < U_{a_i} \). Thus, blocks are always moved toward lower addresses.) For each block \( U_j \) so moved, the final relocation term \( u'_j \) is computed as follows. For the first block \( U_{a'_1} \),

---

* The offset of 1 insures that the first word of the heap is free; this simplifies subsequent garbage collections, c.f. [10].

** Where this is not the case, the necessary adjustments to the algorithm should be obvious.
\[ u''_i = (A'_i - U_{a_i}) + r_i \]

For all other blocks \( U_j \):

\[ u''_j = u'_{j-1} + (V_{j-1} + 1) - U_j \]

(The boundaries \( U_j \) and \( V_j \) are obtained from the information entered in the free list after the initial trace, c.f. Figure 1.) The relocation pair \( \langle \hat{U}_j, u''_j \rangle \) for \( U_j \) is added to the relocation map for the region. Since blocks are moved in order, the relocation pairs are ordered on their first component.

(4) The relocation map is copied from its temporary site in the free buffer to the free space following the packed segment \( S_i \). (It is shown in the Appendix that there is enough free space there to hold the map.)

(5) The region descriptor \( D_i = \langle M'_i, s_i \rangle \) is entered into the region directory.

If \( R_i \) is a trivial region, then no compactification is required. A special descriptor is, however, entered into the region directory to be used in updating pointers. Let \( R_k \) be the most recent nontrivial region. Then \( D_i = \langle Z'_k, 0 \rangle \).

\( Z'_k \) is the address of the relocation pair for the block whose end lies in \( R_i \) (if such a block exists). The second component, 0, is a flag denoting a trivial region.

Additionally, the quantities \( r_i, M'_i \) and \( Z'_i \) (which are needed for computations in the next region) are obtained:

\[ r_i = r_{i-1} \]

\[ M'_i = M'_{i-1} \]

\[ Z'_i = Z'_{i-1} \]

2.3 Updating Pointers

This step adjusts each pointer to point to the final address of the referenced object. Assume that the active pointers can be found and processed each exactly once. By a suitable choice of region size and placement, we can also assume that

(a) the region size is \( 2^p \) for some \( p \)

(b) addresses of all words in a region differ only in their \( p \) least significant bits, i.e., their \( r \) most significant bits are identical.
Let $\Pi$ be a pointer to be updated with initial contents $W$. The $r$ most significant bits of $W$ give the region index $i$ and hence the descriptor $D_i$ for the region in which the referenced object lay before compactification. Two cases arise.

(1) The second component of $D_i$ is non-zero, i.e., $D_i$ is nontrivial. $D_i = (M_i', s_i')$ gives the location and extent of the relocation map $M_i$. A binary search of the map using the $p$ least significant bits of $W$ is performed to obtain the relocation pair $(\hat{U}_j, u''_j)$ for the block referenced by $\Pi$. The updated address is $W'' = W + u''_j$. (Figure 2 shows the computation in the situation shown in Figure 1. Numbers are in octal; a pointer takes 7 bits; word size is $12_8$ bits; the heap starts at 0; the directory starts at $100_8$; region size is $10_8$.)

(2) The second component of $D_i$ is zero. Hence, $D_i$ is trivial and $D_i = (Z_k', 0)$ where $Z_k'$ is the address of the relocation pair $(\hat{U}_k, u'_k)$ for the block $U_k$ whose end lies in $D_i$. Clearly, $\Pi$ points into $U_k$ and $W'' = W + u'_k$.

2.4 Global Compactification

The final step is moving the packed segments of the various regions to form a single contiguous block. To do this, the lower and upper addresses of the packed segments are calculated for each nontrivial region. $A'_i$ is recomputed as $\text{Max}(Z_{i-1}, Z'_i-1)+1$ where $Z_{i-1}$ is obtained from $D_{i-1} = (M'_{i-1}, s_{i-1})$ as $Z'_{i-1} = M'_{i-1} + s_{i-1} - 1$. The upper address, $M'_i-1$, is obtained from $D_i$. Moving the packed segments is then straightforward.

2.5 Discussion

The most expensive operation in compactification is determining for each pointer the final address of the referenced block. Choice of region size as a power of 2 and placement of regions on appropriate address boundaries allows one-step access to the appropriate relocation map through the region directory. If the region size is $2^p$, a relocation map has at most $2^{p-1}$ entries, so a binary search of the map takes at most $p-1$ steps, generally far less. Typical values for $p$ are 10 or less.*

*For example, an IBM 370 with $2^{22}$ words of heap and a 4K entry region directory gives $p = 10$; a PDP 10 with $2^{18}$ words of heap and a 512 word region directory gives $n = 9$. 
The proposed algorithm has the apparent drawback of moving each block twice. However, since the time to move blocks is generally negligible compared to the time spent in the initial trace and in updating pointers, this is not a significant defect. In regard to space, there is a trade-off between the size of the region directory and the size of the space reserved as a temporary site for the relocation maps, i.e., their product must be greater than or equal to half the heap size. Their total space is minimized when each is \((\text{heap size}/2)^{1/2}\).

3. IMPLEMENTATION

3.1 Finding the Pointers

Section 2.3 assumes that all active pointers in the heap can be located during pointer updating. We consider two techniques for doing this: a second trace and an expanded bit map.

3.1.1 Second Trace

As the basic tracing algorithm has been previously discussed, it is necessary to consider only one problem: locating the referenced object during the updating trace. During the updating, all objects are in their interim locations. Consider a pointer containing an address \(W\) referencing some object in the block \(\mathcal{U}_j\) belonging to \(\mathcal{R}_i\). \(W''\) is needed for updating and its computation is discussed in Section 2.3. For tracing, \(W'\) is needed as well. There are three possible implementations.

(A) If there is room in a word, the relocation pairs can be replaced by relocation triples \(\langle \hat{U}_i, u'', u'_i \rangle\). \(W'\) is then \(W + u'_i\).

(B) A third component, \(r_i\) can be added to each region descriptor \(\mathcal{D}_i\). \(W'\) is then \(W'' - r_i\). Often, this will double the size of the region directory.

(C) The first relocation pair of \(\mathcal{M}_i\) can be replaced by \(\langle u'_i, r_i \rangle\). Since \(r_i\) is available, \(W'\) is then \(W'' - r_i\). The first relocation pair is treated as a special case; when it is necessary to compute \(\langle \hat{U}_{a_i}, u'' \rangle\), the following relations are used.

\[
\begin{align*}
    u''_{a_i} &= u'_i + r_i \\
    \hat{U}_{a_i} &= A'_i - A_{a_i} - u'_i
\end{align*}
\]
Trivial regions also require special treatment since the relocation term for the most recent nontrivial region is needed; the modifications are straightforward.

3.1.2 Expanded Bit Map

By using more than one bit to "mark" a word, the initial trace can be used to create a descriptor of the locations of all pointers in the heap. Locating pointers for updating is then performed by a linear sweep of the descriptors. Two cases are common.

(A) Only one pointer can be stored in a word (e.g., the IBM 370). The system is constructed so that if a word contains a pointer, that pointer is in a known position (e.g., right adjusted). Two bits describe each word: a mark bit and a contains-a-pointer bit. The updating step adjusts the contents of words whose pointer bit is on. At that time, the pointers are in their interim locations which must be computed from their original addresses given by the map index of the bits; this computation is straightforward.

(B) Two pointers can be stored in one word (e.g., the DEC PDP 10). The system is constructed so that if a word contains only one pointer, it is a known position (e.g., right adjusted). Two bits describe each word with the code:

- 00 — the word is not marked
- 01 — the word is marked and contains no pointers
- 10 — the word is marked and contains one pointer
- 11 — the word is marked and contains two pointers

The updating step adjusts the contents of words with codes 10 and 11.

3.2 Size of Relocation Pairs and Region Descriptors

An essential assumption of the algorithm is that a relocation pair will fit in one word.* A relocation pair is \( \langle \hat{U}_j, u''_j \rangle \). The final relocation term \( u''_j \) can be as

* More precisely, it would be correct to consider "units of allocation and reclamation" rather than "words." A possible, but expensive, solution is to use N words as the basic unit.
large as the heap size. The relative address $U_j$ is at most as large as a region. The assumption is satisfied whenever the number of bits per word exceeds the sum of $\log_2(\text{heap size}) + \log_2(\text{region size})$. For almost all machines, this is readily satisfied with an acceptable choice of region (and hence directory) size.*

Similar considerations apply to a descriptor $(M'_i, s_i)$, since $M'_i$ may be as large as the heap size while $s_i$ is at most half the region size.

Hence, partitioning the heap into regions serves two complementary functions. It allows fitting a relocation pair into one word and, therefore, the relocation map into the unused portion of the heap. It confines the search for relocation pairs to small regions and therefore shortens it.

3.3 Storage for the Free Buffer

A free buffer for the region directory and the temporary storage of relocation maps has been assumed. Depending on how pointers are updated, this may be stored entirely or in part in the bit map. (Similar considerations apply to the garbage collector's trace stack if one is used.)

3.3.1 Second Trace

The space for the bit map is used only during the trace operations and thus is available during regional compactification for the temporary storage of relocation maps. However, it is not possible to use the space for the region directory as this must be available during the pointer updating trace. Hence, room for the region directory must generally be reserved elsewhere.

A variation on the second trace can sometimes be used to release the bit map. During the updating trace, it is not necessary to mark accessible storage (as in the first trace) but merely to flag updated pointers. Suppose there is available space in words containing pointers but no extra space in non-pointer words (e.g., the IBM 370 and DEC PDP 11). In this case, it is advantageous to reserve a flag bit in each pointer word and use it for the second trace. This frees the bit map storage for the region directory during pointer updating.

*The IBM 370 is a rather extreme example. With 32 bits per word and an address space of $2^{22}$ words, 10 bits remain for $U_j$. The region size must be no more than 1K words, implying a directory of at least 4K words. That is, .1% or more of memory (to be contrasted with the 3% for the bit map)
3.3.2 Expanded Bit Map

(A) One pointer per word.

Of the two garbage collection bits, only the one indicating contains-a-pointer is required during updating. If the bits are stored in two separate arrays, the bit map for the mark bits is unused once the free list has been formed. That storage may be used for both the temporary storage of relocation pairs and the region directory. As the bit map contains (heap size/bits per word) words, and the storage required for pairs and directory can be as small as $(2 \cdot \text{heap size})^{1/2}$ words (c.f. Section 2.5), there is generally a surplus of space. This is best used by expanding the directory, thereby reducing the region size and, hence, the search* as far as practical.

(B) Two pointers per word.

Neither bit map is unused during regional compactification. As the free buffer must be reserved, it is generally desirable to minimize its size by choosing the region size to be the least $p$ such that $2^p \geq (2 \cdot \text{heap size})^{1/2}$.

3.4 Discussion

Three sorts of operations are used in the proposed algorithm: tracing, updating pointers, and moving blocks. Their relative expense depends on the number of blocks per region and the ratio of pointer to non-pointer fields. Under reasonable assumptions, trace time and update time have the same order of magnitude while block moving time is far less. Replacing the second trace with an extended bit map saves on the order of one third of the total time.

To a first approximation, the choice can be made as follows. In a small address space, reserving space for the second bit map is unacceptable so a second trace is required; in a large address space, storage is generally less tight and the time saving (proportional to heap size) more important.

*Note that the region size is then close to the number of bits per word, making the search very short. In this case, a linear search, requiring less set-up time, may be faster than a binary search.
Less obvious considerations are as follows.

(1) When two traces are used, their tasks are different and extra code is required. Space for the code may partially offset the saving in having only one bit map, particularly if trace routines for many data types are used.

(2) In a virtual memory system, tracing is relatively expensive due to page faults so that avoiding a second trace is more significant.

(3) When a word can hold only one pointer and a double bit map is used, the re-use of one map as the free buffer implies that no additional storage is needed for the free buffer.
REFERENCES


5. Goyer, P. A Garbage Collector to be Implemented on a CDC 3100, in *Algol 68 Implementation*.


10. Wegbreit, B. A Generalized Compactifying Garbage Collector, 

11. Wodon, P. L. Methods of Garbage Collection for Algol 68, in Algol 68 
Implementation.
APPENDIX

We prove claims (made in Section 2.2) that during regional compactification free blocks can always be moved toward lower addresses and there is always room to hold the relocation map for a region in the free space following the packed segment. The following assumptions are employed.

- The unit of allocation and reclamation is the word; hence any free block is at least one word long,
- A relocation pair can be stored in a word,
- A word has been reserved at the beginning of the heap and at the end of the heap.

Let $a_i$ and $b_i$ denote the indices of the first block $U_{a_i}$ and last block $U_{b_i}$ of region $R_i$. Let $B$ denote the low address of the heap.

**Proposition**

(1) $A'_i \leq U_{a_i}$

(2) $Z'_i \leq V_{b_i} + 1$

**Proof**

(I) We show that for all $i$, (1) implies (2). $Z'_i - A'_i + 1$ is the sum of the sizes of the relocation map $M_i$ and of all used blocks belonging to $R_i$. So, $Z'_i - A'_i + 1 = b_i + \sum_{j=a_i}^{b_i} (V_j - U_j + 1)$. $V_{b_i} - U_{a_i} + 1$ is the size of the space originally occupied by the used blocks belonging to $R_i$ and the free blocks in between. Thus it is at least as large as the sum of the sizes of used blocks belonging to $R_i$ plus the number of free blocks in between. Hence, $V_{b_i} - U_{a_i} + 1 \geq \sum_{j=a_i}^{b_i} (V_j - U_j + 1) + s_i - 1$. Combining the two relations, $V_{b_i} - Z'_i + 1 \geq U_{a_i} - A'_i$. Thus, $A'_i \leq U_{a_i}$ implies $Z'_i \leq V_{b_i} + 1$.

(II) We show (1) by induction on $i$. For $i=1$, $a_i=1$. $A'_1 = B + 1$ by definition. $U_1 \geq B + 1$ since a word is reserved at the start of the heap. Hence, $A'_1 \leq U_1$. For $i > 1$, we may assume that (1) and hence (2) are true up to $i-1$. Then, $Z'_{i-1} \leq V_{b_{i-1}} + 1$. The first block $U_{a_i}$ belonging to $R_i$ follows the last block $U_{b_{i-1}}$ of $R_{i-1}$ with at least one word in between. Hence, $U_{a_i} > V_{b_{i-1}} + 1$, so $U_{a_i} \geq Z'_{i-1} + 1$. As the algorithm sets $A'_i = \max(A'_1, Z'_{i-1} + 1)$, it follows that $U_{a_i} \geq A'_i$. $\square$

Note that (1) insures that blocks are always moved toward lower addresses. That there is always room for the relocation map follows from (2) and the reservation of a word at the end of the heap. The need for the reserved word would be illustrated by Figure 1 had the heap ended with region 5. (While it is possible to modify the algorithm to work without the reserved word, the modified version is more complex.)