Comparison of Compacting Algorithms for Garbage Collection

Ahmed Hussein
Agenda

- Compaction..What is that?
- Presenting four different algorithms
  - Lisp2
  - Table Compactors
  - Morris
  - Jonkers
Overview

Memory

O1  O2  O3  O4  O5  O6  O7  O8  O9
Phase I: Marking

Memory

O1 O2 O3 O4
O5 O6
O7 O8 O9
Phase2..Collecting

Memory
Phase 2: Collecting Memory

Tuesday, February 7, 12
Phase 3..Compaction

Memory

O1
O3
O5
O6
O7
O9
O10
Phase 3..Compaction
Phase3..Compaction
Phase 3.. Compaction

Memory

O1  O3  O5  O9

O7  O6

O10
Phase 3.. Compaction

Memory

O1  O3  O5  O9

O7

O6

O10
Phase 3.. Compaction
Object Model

Size c

npc

data

Size
Lisp2

Memory

Size c1

npc

Size c2

npc
Lisp2

Compacting

Memory

Size c1

npc

Size c2

npc
Lisp2

Compacting

Memory

Size c1

npc

Size c2

npc
Lisp2

Compacting

Memory

Size c1

npc

Size c2

npc
Lisp2

Size c

npc
Lisp2

Address
Size cl
npc
Lisp2

Pass 1

\[ a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \]
Lisp2

Pass 1
Lisp2

Pass 1
Lisp2

Pass 1
Lisp2

Pass 1
Lisp2

Pass 1

a0

a1

a1'

a2

a2' = a1' + size

a3

a4

a4' = a2' + size

a5
Lisp2

Pass2
Lisp2

Pass2

Diagram showing nodes labeled a1, a2, and a4 connected by arrows.
Lisp2

Pass2

\[ a_1 \]
\[ a_1' = a_1' + \text{size } a_1 \]
\[ a_2 \]
\[ a_2' = \]

\[ a_4 \]
Lisp2

Pass2

\[ a_1' \]
\[ a_2' = a_1' + \text{size } a_1 \]
\[ a_4' = a_2' + \text{size } a_2 \]
Lisp2

Pass2
Lisp2

Pass3

\[
\begin{align*}
\text{a1} & \quad \text{a1'} \\
\text{a2} & \quad \text{a2'} = \text{a1'} + \text{size} \\
\text{a4} & \quad \text{a4'} = \text{a2'} + \text{size}
\end{align*}
\]
Pass3

\[ a_1 = a_1' + \text{size} \]

\[ a_2 = a_2' + \text{size} \]

\[ a_4 = a_4' + \text{size} \]
Lisp2

Pass3
Pass3

Lisp2
Lisp2

Pass3

\[ a_1 \]
\[ a_1' \]
\[ a_2' = a_1' + \text{size} \]
\[ a_4' = a_2' + \text{size} \]
Lisp2

Pass 3

\[
\begin{array}{ccc}
  a_1' & a_2' & a_4' \\
  a_1' & a_2' = a_1' + \text{size} & a_4' = a_2' + \text{size}
\end{array}
\]
Lisp2

Pass3

\[
\begin{align*}
&\text{a1} \\
&\quad \text{a1'} \\
&\quad \text{a2'} \\
&\quad \text{a2'} = \text{a1'} + \text{size} \\
&\quad \text{a4'} \\
&\quad \text{a4'} = \text{a2'} + \text{size}
\end{align*}
\]
Lisp2

Pass3

\[
a1' = a1' + \text{size} \\
a2' = a2' + \text{size} \\
a4' = a4' + \text{size}
\]
Lisp2

Pass3

\[ a_1' \]
\[ a_2' = a_1' + \text{size} \]
\[ a_4' = a_2' + \text{size} \]
Lisp2

Pass3

\begin{align*}
\text{a1}' &= a1' \\
\text{a2}' &= a2' \\
\text{a3}' &= a3' + \text{size} \\
\text{a4}' &= a4' + \text{size}
\end{align*}
Lisp2..Final
Lisp2 .. Summary

- Requires 1 extra word in each object for temp pointer. (even when the object is not live)

- Compaction is done in 3 phases:
  1. Traverse the objects, sorted by address
     - Compute new address of each live object
     - free_ptr=0; free_ptr+=free_ptr+size of live object
  2. Update Pointer fields.
  3. Sliding Compaction
Table Compactors

• We need to save the overhead due to temp pointers.
• Using inactive cells to store readjustments.
# Break Table

### Phase I

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>100</th>
<th>300</th>
<th>950</th>
<th>1200</th>
<th>1600</th>
<th>1999</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(100,100)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Tuesday, February 7, 12*
Break Table

Phase I
# Break Table

## Phase I

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>100</th>
<th>300</th>
<th>950</th>
<th>1200</th>
<th>1600</th>
<th>1999</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1999</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1999</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1999</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1999</td>
</tr>
</tbody>
</table>

- (100,100)
- (950,750)
- (950,750)
- (100,100)
- (1600,1150)
Break Table

- Rolling back causes it to become unsorted.
- Need another phase just to sort the BT.
Break Table

- Phase 3 to fix the pointers.
  1. Search through the BT table and determine the adjacent pairs \((a, s)\) and \((a', s')\) such that \(a \leq p < a'\)
  2. readjusted value should be \(p - s\).
Break Table .. Cost

- Phase 1: linear
- Phase 2: nlogn
- Phase 3: nlogn

- We can enhance the last phase by constructing a hash if we have enough space.

- Other suggestions to keep a linked list in holes and update pointers before moving objects.
Problem .. revisited

- It is clear from the previous 2 algorithms that updating pointers is bottleneck.
Threading

A

B

C

Info

P
Threading

A

B

C

Info

P

Info

A

B

C

Info

P
Threading

- After calculating the new address of P we can traverse the list and fix all the pointers to point to the new address of P.
Jonker Algorithm
Jonker Algorithm

First Path
Jonker Algorithm

First Path
Jonker Algorithm

First Path

\[ p' = \text{nextFree} \]
Jonker Algorithm
Jonker Algorithm

Second Path
Jonker Algorithm

Second Path
Analysis of Threaded

- Each object is touched three times.

- Space:
  - Jonker, no space required but each node has a pointer-sized header.
  
- Morris
  
  - 2 tag bits per field, 0 inactive, 1 pointer, 2 swapped pointer, 3 non pointer.
  
  - Could be improved by merging marking phase with first phase.
Threaded..Analysis

- Compact tables touch every object only twice.
Compaction Summary

• Suits smaller physical memory. Semi-Space requires double the memory space.

• For long lived objects, the heap becomes similar to “generational collector”.

• Improve locality.

• Other algorithms have only one path.
How to Compare

- Variable sized objects?
- Directions?
- Have to tag pointer data?
- Time and Space Performance.
Time Comparison

Fig. 8. Time comparisons for the four compactors.