Comparison of Compacting Algorithms for Garbage Collection

By Jacques Cohen & Alexandru Nicolau

Presented by Sam Harwell
Mark/Sweep/Compact Collection

• Garbage collection is performed in two stages
  – Identify objects in memory that are live (*mark*)
  – Make dead-object memory locations available to the allocator by *compacting* live objects

• We are focusing on the compaction algorithm
Allocator Fundamentals

• Memory is allocated for cells:
  – Variable sized
  – Each may hold pointers and/or data

Incoming pointers may only point to the first element in the cell

A basic cell:

<table>
<thead>
<tr>
<th>reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>size c</td>
</tr>
<tr>
<td>npc</td>
</tr>
</tbody>
</table>

Pointers
Data
Algorithms

• Classical
  – Lisp 2
    • D.E. Knuth, 1973
  – Table Compactor
    • Modified algorithm based on Waite and Haddon, 1967

• Modern (at the time)
  – Morris’ algorithm, 1978
  – Jonkers’ algorithm, 1979
Lisp 2

• Requires additional space in each cell for a pointer

• Three passes:
  – Compute new address of each active cell
  – Update pointer fields of each active cell
  – Relocate active cells
Lisp 2 Compactor

Initially:

After pass 1:

After pass 2:

After pass 3:
Table Compactor

- Stores relocation data in garbage cells
- Fundamental: cells are moved forward by the total size of holes preceding the cell
- Pointers are updated similarly
Threading

- An elegant way to answer, “Where are all the pointers to cell X?”
  - Threading the root:
Threading

• An elegant way to answer, “Where are all the pointers to cell X?”
Morris’ Algorithm

- Three-passes: one forward and two backward
- Two tag bits per field overhead (one if single cells cannot hold both pointers and data)
Morris’ Algorithm

Initial:

Cell c is reached:

Update back ptr to c:

Move cell containing c:

Updated ptr to self:

Finished:
Jonkers’ Algorithm

• Improved on Morris’:
  – Only two passes, both forward
  – One bit per cell overhead

• Added assumptions:
  – No pointer-to-members
  – A cell containing data is always large enough to store a pointer
Jonkers’ Algorithm

Initially:

Cell c reached in pass 1:

Threaded lists updated:

Processed self reference:

Back pointers to c threaded:

Cells below c compacted:

Threaded list updated:

Compacted:
Time-Formulas

• Create optimized versions of each algorithm
• Describe each procedure with a formula from the type and number of operations performed
• Replace unknowns in the formula with machine-specific constants, leaving the following variables:
  - $\alpha$: Marked cell ratio $(NMC/NC)$
  - $\beta$: Live pointer ratio $(NAP-1)/(NPC*NMC)$
## Feature Comparison

<table>
<thead>
<tr>
<th></th>
<th>Lisp 2</th>
<th>Table compact</th>
<th>Morris’</th>
<th>Jonkers’</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>1 pointer/cell</td>
<td>None</td>
<td>2 bits/field</td>
<td>1 bit/cell</td>
</tr>
<tr>
<td><strong>Passes</strong></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>Member ptr</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Performance Results

Fig. 8. Time comparisons for the four compactors.
Effects of Increased Sorting in the Table Compactor
Effects of Cell Size

\[ NC = 500,000 / \text{Size } c \]

\[ a = 0.5 \]

\[ \beta = 0.9 \]

\[ NPC = 5 \]