Generational Garbage Collection

Mirko Jerrentrup, Jerrentrup@interactive-software.de
Overview

- motivation
- at a glance
- issues
- problems and limitations
- conclusion
Motivation

„classical“, e.g. copying GC:

• no improvement of locality

• repeated handling of long-lived objects
The weak generational hypothesis

„Most objects die young.“

- partition objects into *generations*
- special handling of *young* objects:
  - reduced pause times
  - better collection efficiency

[Ungar, 1984]
Generational garbage collection

- objects partitioned into *multiple generations*

- young generation *frequently collected* (minor collection)

- old generation *seldomly collected* (major collection)

- surviving young objects *promoted* into old generation
Example: minor collection

- **root set**
- **old generation**
- **young generation**
- **root set for minor collection**
- **Inter-generational pointer**

- **live nodes**
- „garbage“ nodes
Issues in generational garbage collection

- Promotion
- Heap organization
- Scheduling
- Inter-generational pointers

generational garbage collection
generational garbage collection

- promotion
- scheduling
- heap organization
- inter-generational pointers
Promotion

early promotion

better

long-living objects

worse

short-living objects

worse

better

late promotion

time
Promotion policies

- fixed
- adaptive

Appel’s collector for SML/NJ
demographic feedback-mediated tenuring
Appel’s collector for SML/NJ (1)

**Idea:** manage promotion rates by fixing heap occupancy of young objects

- two generations,
  *very large young generation*

- major collections only if old objects occupy *half size of heap*

- precondition: *contiguous heap*
Appel’s collector for SML/NJ (2)

- old  reserve  free
- old  reserve  new
- old  svr  reserve  free

- half heap size
- old  svr  free
- free  svr  old  free
- old  reserve  free

- equal size
Idea: promote only when necessary to hold maximum pause time

• only promote if \textit{pause time} will be \textit{acceptable}

• generate \textit{space-age table}

• promote \textit{only as many objects} to make pause time acceptable

[Ungar, Jackson 1992]
Demographic feedback-mediated tenuring (2)

- **young generation**
  - free
  - survivors

- **maximum acceptable pause time**

  - No promotion at next collection
  - Promotion of 150 bytes "oldest" objects

  - Size of objects in age group
    - Age 1: 300 bytes
    - Age 2: 200 bytes
    - Age 3: 100 bytes

- Number of survived collections

[Content from Ungar, Jackson 1992]
generational garbage collection

- promotion
- scheduling
- heap organization
- inter-generational pointers
Determination of object's generation

- copying collectors: *subheaps* for generation
  
  contiguous heaps $\rightarrow$ object address
  non-contiguous heaps $\rightarrow$ header field or page-table

- non-copying collectors $\rightarrow$ header field
Heap organization schemes

- creation space
- "high water mark" bucket system
Goals:

- no large semi-spaces
- improve locality

Organization of a generation:

- (small) *aging area* in two semi-spaces
- (large) *new object area* in one *creation space*

[Ungar, 1984]
Creation space (2)

[Diagram showing generation occupancy, creation space, aging semi-spaces, and time with gc pause]

[Ungar, 1984]
Goals:

• avoid age field in object header
• adaptive promotion threshold

Organization of generations:

• creation and aging spaces per generation
• two buckets per generation
• creation space partly holds first bucket
• high water mark separates buckets

[Wilson, Moher 1989]
"High water mark" bucket system (2)

Wilson, Moher 1989

heap occupancy

younger generation

next generation

bucket 1

high water mark

bucket 2

time

[Wilson, Moher 1989]
„high water mark“ effect:

- objects from bucket 2 are promoted into older generation
- objects from bucket 1 are stored in bucket 2
- „high water mark“ position determines promotion threshold
- promotion threshold between 1 and 2
generational garbage collection

- promotion
- heap organization
- scheduling
- inter-generational pointers
Perform collection when:

- pause is not interruptive
- large amount of garbage can be expected
Key (large) objects:

- objects whose "death" produces much garbage (root of a large tree etc.)
- exclude key objects from generational scheme
- store key objects "descendants" in special large object area
- reclaiming of key objects trigger collection in key area

[Hayes 1991]
Efficient collections – key objects (1)
Efficient collections – key objects (2)

- root set
- key objects
- young generation
- keyed area

[Hayes 1991]
generational garbage collection

- promotion
- heap organization
- scheduling
- inter-generational pointers
Inter-generational pointers

Issues:

• detecting creation
• including into root set upon collection
• cost of detection / inclusion
Inter-generational pointer handling

- detection / storing of inter-generational pointers
- detecting individual pointers
  - pointer indirection
  - remembered sets
- marking pointer-containing areas
  - page / word / card marking
Detection of inter-generational pointers:

- trapping pointer stores
- only necessary to check objects in old generation(s)
- generally: only non-initializing stores

write-barrier
**Idea:** indirect pointers through an *entry table*

[Image of a diagram showing different generations of pointers with an entry table.]

[Liebermann, Hewitt 1983]
**Idea:** indirect pointers through an *entry table*

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>trapping stores (by mutator)</td>
<td>small constant for <em>every</em> store</td>
<td>O(#stores), small constant</td>
</tr>
<tr>
<td>inter-generational pointer-handling at collection time</td>
<td>O(#stores)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

All pointers in the entry table are added to the root set.

[Liebermann, Hewitt 1983]
Idea: *remember* objects with old-young pointers

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>trapping stores</td>
<td>small constant for</td>
<td>O(# pointer-containers</td>
</tr>
<tr>
<td>(by mutator)</td>
<td>every store</td>
<td>)</td>
</tr>
<tr>
<td>inter-generational pointer-handling at collection time</td>
<td>O(# pointer-containers )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

[Ungar 1984]
**Idea:** *mark* (virtual) memory *pages* containing objects with inter-generational pointers

- hardware / virtual memory management support
- only slight overhead for write barrier
- problems intercepting VM signals
- large pages $\rightarrow$ high collection cost

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>trapping stores (by mutator)</td>
<td>small constant for <em>every</em> store</td>
<td>N/A</td>
</tr>
<tr>
<td>inter-generational pointer-handling at collection time</td>
<td>$O(# \text{i.g.p.} \cdot \text{size of page})$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

pages are scanned for pointers

[Shaw 1988]
[Moon 1984]
**Idea:** don’t mark to-large pages or to-small words

- divide address space into *cards* (~128 bytes)
- lower collection cost if card size is near object size
- very low cost for write barrier: 2-3 instructions

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>trapping stores (by mutator)</td>
<td>small constant for <em>every</em> store</td>
<td>small portion of heap</td>
</tr>
<tr>
<td>inter-generational pointer-handling at collection time</td>
<td>$O(# \text{i.g.p.} \cdot \text{size of card})$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

(cards are scanned for pointers)

[Wilson, Moher 1989]
[Sobalvarro 1988]
Remembered sets vs. card marking

Remembered sets (with sequential store buffers):

- no scanning upon collection ✓
- collection overhead $O(\#\text{pointer stores})$ ✗
- duplicates in sequential store buffers ✗
- small overhead for write barrier (2-3 instructions) ✓
Remembered sets vs. card marking

Card marking

- scanning upon collection \( \times \)

- collection overhead \( O(#\text{inter-gen. pointers}) \) \( \checkmark \)

- small overhead for write barrier (2-3 instructions) \( \checkmark \)

- long-lived object’s cards must be scanned repeatedly \( \times \)
hybrid card marking / remembered set GC

• write barrier like card-marking

• after collection, old-young-pointers are added to remembered set

[Hosking, Hudson 1993]
Problems and limitations

heuristic failure:

• cluster of long-lived objects
  „pig in the snake“

• small heap-allocated objects

• large root sets
Conclusion

• improvement, if assumptions hold

• highly variable
Thank you !