An Efficient Non-Moving Garbage Collector for Functional Languages

Katsuhiro Ueno
Atsushi Ohori
Toshiaki Otomo

RIEC, Tohoku University

ICFP 2011,
20 May 2011
Copying Collectors
Mark-and-Sweep Collectors

Copying Collectors
Mark-and-Sweep Collectors

Copying Collectors

Good!

fast allocation by "bump pointer"
Mark-and-Sweep Collectors

Copying Collectors

- Good!
- fast allocation by "bump pointer"
- Good!
- $O(\text{numLives})$ collection cost
Mark-and-Sweep Collectors

Copying Collectors

Good!

Good!

Good!

fast allocation
by "bump pointer"

$O(numLives)$
collection cost

many short-lived objects
frequent allocations
and collections

Functional Programs
Mark-and-Sweep Collectors

- many short-lived objects
- frequent allocations and collections

Functional Programs

Copying Collectors

- fast allocation by "bump pointer"
- $O(\text{numLives})$ collection cost
- object moving
- good for generational collection

Good!
Mark-and-Sweep Collectors

- fragmentation
- slow allocation

Copying Collectors

- fast allocation by "bump pointer"
- \(O(numLives)\) collection cost
- object moving
- good for generational collection

Functional Programs

- many short-lived objects
- frequent allocations and collections
Mark-and-Sweep Collectors

- fragmentation
- slow allocation
- \(O(\text{heapSize})\) collection cost

Copying Collectors

- fast allocation by "bump pointer"
- \(O(\text{numLives})\) collection cost
- object moving
- good for generational collection

many short-lived objects
frequent allocations and collections

Functional Programs
Mark-and-Sweep Collectors

Bad...

fragmentation
slow allocation

Bad...

$O(\text{heapSize})$
collection cost

Bad...

non-moving

no way of
generational
collection

"Good!"

Copying Collectors

Good!

fast allocation
by "bump pointer"

Good!

$O(\text{numLives})$
collection cost

Good!

object moving

good for
generational
collection

"Good!"

many short-lived objects
frequent allocations
and collections

Functional Programs
Mark-and-Sweep Collectors

- avoid in practice
  - fragmentation
  - slow allocation

Copying Collectors

- good for generational collection
- many short-lived objects frequent allocations and collections

Non-moving
- no way of generational collection

Functional Programs

- fast allocation by "bump pointer"
- $O(\text{heapSize})$ collection cost
- $O(\text{numLives})$ collection cost
Mark-and-Sweep Collectors

- avoid in practice
  - fragmentation
  - slow allocation

$O(\text{heapSize})$ $O(\text{numLives})$

collection cost

non-moving

no way of generational collection

many short-lived objects frequent allocations and collections

Functional Programs

Copying Collectors

- fast allocation by "bump pointer"

$O(\text{numLives})$
collection cost

object moving

good for generational collection
Mark-and-Sweep Collectors

- avoid in practice
- fragmentation
- slow allocation

Copying Collectors

- fast allocation by "bump pointer"
- $O(\text{numLives})$ collection cost

non-moving

- non-moving generational collection

Functional Programs

- any short-lived objects frequent allocations and collections
- good for generational collection
Mark-and-Sweep Collectors

- avoid in practice
- fragmentation
- slow allocation

$O(\text{heapSize})$, $O(\text{numLives})$

collection cost

non-moving
don't hold

non-moving
generational collection

any short-lived objects
frequent allocations
and collections

Functional Programs

Copying Collectors

- Good!
- fast allocation
  by "bump pointer"

$O(\text{numLives})$
collection cost

- Good!
object moving

- Good!
good for
generational collection
Mark-and-Sweep Collectors

- avoid in practice
- fragmentation
- slow allocation

\(O(\text{heapSize})\) and \(O(\text{numLives})\) collection cost

- non-moving

non-moving generational collection

Copying Collectors

- Good!
- fast allocation by "bump pointer"
- Good!
- fragmentation-free by definition

\(O(\text{numLives})\) collection cost

- Good!
- object moving

- Good!
- good for generational collection

any short-lived objects and frequent allocations

Functional Programs
Mark-and-Sweep Collectors

- avoid in practice
- fragmentation
- slow allocation

Copying Collectors

- fast allocation by "bump pointer"
- fragmentation-free by definition
- collection cost
- keep good locality
- object moving

non-moving

non-moving generational collection

any short-lived objects frequent allocations and collections

Functional Programs
Mark-and-Sweep Collectors

- avoid in practice
  - fragmentation
  - slow allocation

Copying Collectors

- fast allocation by "bump pointer"
- fragmentation-free by definition
- keep good locality

non-moving

non-moving generational collection

any short-lived objects frequent allocations and collections

Functional Programs

object moving

good for generational collection
Mark-and-Sweep Collectors

avoid in practice

non-moving itself is also a good property!

non-moving

non-moving generational collection

Copying Collectors

fast allocation by "bump pointer"

fragmentation-free by definition

keep good locality

object moving

good for generational collection

Functional Programs

any short-lived objects frequent allocations and collections
Viva! Non-moving

Very easy to share heap objects.
- No need to do *pinning* when interacting with C and other languages.

Very easy to maximize concurrency.
- No need to stop threads for managing shared objects.

These are *free* when you choose non-moving!
Mark-and-Sweep Collectors
- avoid in practice
  - fragmentation
  - slow allocation

Copying Collectors
- fast allocation by "bump pointer"
- \( O(\text{numLives}) \) collection cost

Non-moving
- non-moving generational collection
- good for generational collection

Functional Programs
- any short-lived objects frequent allocations and collections
Mark-and-Sweep Collectors
- avoid in practice
- fragmentation
- slow allocation

Copying Collectors
- fast allocation by "bump pointer"
- \( O(\text{numLives}) \) collection cost

Functional Programs
- any short-lived objects frequent allocations and collections

- non-moving
- non-moving generational collection

- object moving
- good for generational collection
Mark-and-Sweep Collectors

- avoid in practice
- fragmentation
- slow allocation

Bad...

O(\text{heapSize})

collection cost

Bad...

non-moving generational collection

Good!

collection cost

O(\text{numLives})

any short-lived objects and collections

Good!

Functional Programs

choice

non-moving

Good!

object moving

Good!

good for generational collection

Copying Collectors

fast allocation by "bump pointer"

O(\text{numLives})

Good!
The topic of this talk

We propose a non-moving GC which is as efficient as Cheney’s copying GC.
Weakness of mark-and-sweep GC

- Fragmentation, and slow allocation
- High sweep cost \(O(\text{heapSize})\)
- No known method for extending it to non-moving generational GC

We choose a well-known idea of bitmap marking as our start point to overcome these weaknesses...
Strategy

But bitmap marking strategy alone does not yield an efficient GC.

We re-organize the bitmap marking with:
- fragmentation avoiding heap organization
- tree structured bitmap
- automatic heap size adjustment
- non-moving generational extension

with a series of optimized bit manipulation algorithms.
Fragmentation

Varied size objects incur fragmentation.

If all objects were same size, heap could be a fixed size array without incurring fragmentation.
Avoid fragmentation

Separate the heap for each object size.
Avoid fragmentation

Separate the heap for each object size.

\[ H_3 \]

8 bytes/block

\[ H_4 \]

16 bytes/block

\[ H_5 \]

32 bytes/block

\[ H_6 \]

64 bytes/block
Tree-structured bitmaps

For fast allocation and collection

```
1 1 ... 1 0 1 1 ... ... 1 0 1 1 ... ... 0
```

```
... ... ... ... ... ... ...
```
Tree-structured bitmaps

For fast allocation and collection
Tree-structured bitmaps

For fast allocation and collection

[Diagram illustrating tree-structured bitmaps with bit pointers and allocation pointers]
Tree-structured bitmaps

For fast allocation and collection
Tree-structured bitmaps

For fast allocation and collection
Tree-structured bitmaps

For fast allocation and collection

```
0 0 … 0
1 0 1 … 1 1 0 … 0
1 1 … 1 0 1 1 … 1 0 1 1 … 0
```

…

```
Tree-structured bitmaps

For fast allocation and collection
Tree-structured bitmaps

For fast allocation and collection
Tree-structured bitmaps

For fast allocation and collection
Tree-structured bitmaps

For fast allocation and collection
Tree-structured bitmaps

For fast allocation and collection

```
0 0 ...

0 1 0 ...

0 1 ...

... ...

... ...
```
Sub-heap size adjustment

<table>
<thead>
<tr>
<th>Heap</th>
<th>$H_3$</th>
<th>$H_4$</th>
<th>$H_5$</th>
<th>$H_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8b/blk</td>
<td>16b/blk</td>
<td>32b/blk</td>
<td>64b/blk</td>
</tr>
</tbody>
</table>

Which layout is appropriate? 
... cannot determine it in advance.
Sub-heap size adjustment

Heap

\[ H_3 \quad \begin{array}{c|c|c} \hline 1 & 0 & 1 \hline \end{array} \ldots \quad \begin{array}{c|c|c} \hline 0 & 0 & 1 \hline \end{array} \ldots \]

8 bytes/block

\[ H_4 \quad \begin{array}{c|c|c} \hline 1 & 1 & \vdots \hline \end{array} \ldots \quad \begin{array}{c|c|c} \hline 0 & 1 & \vdots \hline \end{array} \ldots \quad \begin{array}{c|c|c} \hline 1 & 0 & \vdots \hline \end{array} \ldots \quad \begin{array}{c|c|c} \hline 0 & 0 & \vdots \hline \end{array} \ldots \]

16 bytes/block

\[ H_5 \quad \begin{array}{c|c|c} \hline 1 & \vdots \hline \end{array} \ldots \quad \begin{array}{c|c|c} \hline 0 & \vdots \hline \end{array} \ldots \]

32 bytes/block

\[ H_6 \quad \begin{array}{c|c|c} \hline 1 & \vdots \hline \end{array} \]

64 bytes/block

\[ \vdots \]
Generational extension

Refining the idea of partial GC (Demers et al 1990)

- generations = disjoint sets of objects
- sets of objects = bitmaps

\[ \mathcal{G}_3 = \{ O_3, O_4, \ldots \} \]
\[ \mathcal{G}_2 = \{ O_2, O_6, \ldots \} \]
\[ \mathcal{G}_1 = \{ O_1, O_5, O_7, O_8, \ldots \} \]

block array

|   | O₁ | O₂ | ⋯ | O₃ | O₄ | O₅ | ⋯ | ⋯ | O₆ | O₇ | O₈ | ⋯ |
**Generational extension**

Refining the idea of partial GC (Demers et al 1990)

- generations = disjoint sets of objects
- sets of objects = bitmaps

\[ G_3 = \begin{array}{cccccccccc}
0 & 0 & \cdots & 1 & 0 & 1 & 0 & \cdots & \cdots & 0 & 0 & 0 & 0 & 0 & \cdots
\end{array} \]

\[ G_3 \cup G_2 = \begin{array}{cccccccccc}
0 & 1 & \cdots & 1 & 0 & 1 & 0 & \cdots & \cdots & 1 & 0 & 0 & 0 & 0 & \cdots
\end{array} \]

\[ G_3 \cup G_2 \cup G_1 = \begin{array}{cccccccccc}
1 & 1 & \cdots & 1 & 0 & 1 & 1 & \cdots & \cdots & 1 & 0 & 1 & 1 & \cdots
\end{array} \]

block array

\[ O_1 \quad O_2 \quad \cdots \quad O_3 \quad O_4 \quad O_5 \quad \cdots \quad \cdots \quad O_6 \quad O_7 \quad O_8 \quad \cdots \]
Performance evaluation

We compared

- our method,
- Cheney’s copying collector,
- our method with 2 generations, and
- generational copying (based on Reppy 1994)

by extensive benchmarks on our SML# compiler.
### Memory usage

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size (MB)</th>
<th>our method</th>
<th>copying</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>live</td>
<td>occ.</td>
</tr>
<tr>
<td>count_graphs</td>
<td>2</td>
<td>6.02</td>
<td>55.4</td>
</tr>
<tr>
<td>cpstak</td>
<td>2</td>
<td>5.56</td>
<td>51.6</td>
</tr>
<tr>
<td>knuth_bendix</td>
<td>12</td>
<td>10.45</td>
<td>61.1</td>
</tr>
<tr>
<td>ratio_regions</td>
<td>20</td>
<td>11.97</td>
<td>62.3</td>
</tr>
<tr>
<td>gcbench</td>
<td>65</td>
<td>10.61</td>
<td>65.9</td>
</tr>
<tr>
<td>perm9</td>
<td>190</td>
<td>22.36</td>
<td>57.6</td>
</tr>
</tbody>
</table>

- **live**: ratio of survivals against GC
- **occ.**: ratio of memory amount filled with data
Benchmark: GC time

![Graphs showing the performance of different garbage collection algorithms (cheney gc, bitmap gc, copy(2g) gc, bitmap(gen) gc) across various heap sizes (1 MB to 8 MB, 8 MB to 24 MB, and 60 MB to 200 MB). The graphs compare the GC time (in seconds) to the heap size (in MB) for each algorithm. The graphs indicate that cheney gc generally has the lowest GC time, followed by bitmap gc, copy(2g) gc, and bitmap(gen) gc. The algorithms are compared across different benchmarks: knuth_bendix, count_graphs, and gcbench.]
**Benchmark: mutator time**

- **Mutator Time [sec]**
  - cheney mutator
  - bitmap mutator
  - copy(2g) mutator
  - bitmap(gen) mutator

- **Heap Size [MB]**
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8

- **Mutator Time [sec]**
  - 0.45
  - 0.5
  - 0.55
  - 0.6
  - 0.65
  - 0.7
  - 8
  - 10
  - 12
  - 14
  - 16
  - 18
  - 20
  - 22
  - 24

- **Heap Size [MB]**
  - 60
  - 80
  - 100
  - 120
  - 140
  - 160
  - 180
  - 200

**Graphs:**

- knuth_bendix
- count_graphs
- gcbench
Ratio of total time (ours / copying)

Minimum heap size x 2

[Bar chart showing ratios for various benchmarks]
Ratio of total time (ours / copying)

Minimum heap size x 3
Ratio of total time (ours / copying)

Minimum heap size x 4

19 / 20
Ratio of total time (ours / copying)

Minimum heap size x 5

Ratio bar chart showing the ratio of total time for various benchmarks, with the x-axis listing benchmarks such as 'barnes-hut', 'count-graphs', 'diviter', 'fft', 'life', 'logic', 'mandelbrot', 'puzzle', 'ray', 'simple', 'boyer', 'knuth-bendix', 'mlyacc', 'ratio-regions', 'smilboyer', 'tsp', 'viliw', 'gcbench', and 'perm9'. The y-axis represents the ratio, with values ranging from 0 to 1.2. The data shows a line at 1, indicating a 1:1 ratio, with many benchmarks showing values close to 1, indicating similar performance. The graph highlights the performance comparison between different benchmarks under the minimum heap size condition.
Conclusion

We have developed an efficient non-moving GC.

- avoid fragmentation by separating the heap.
- fast allocation and GC through bitmap trees.
- generational GC through multiple bitmaps.

A viable alternative to copying GC for functional languages.

Further Development

non-moving concurrent GC