Garbage Collection: Tracing or Reference Counting?

Peter Libič
Department of Distributed and Dependable Systems

http://d3s.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
(Short) update on my “progress” on garbage collection simulation/modeling work

Performance analysis and optimizations of reference counting GC
  • Comparison with tracing

GC Simulation/Modeling Update

• Last time:
  • Presentation of methodology I am using
  • How does a typical experiment look like?
  • Some experiment results
    – Very little improvement
  • Light at the end of a tunnel?
GC Simulation/Modeling Update

- Last time:
  - Presentation of methodology I am using
  - How does a typical experiment look like?
  - Some experiment results
    - Very little improvement
  - Light at the end of a tunnel?
    - It was a train ...
Current Status

X axis – Heap configuration ID
Y axis – Full GC counts

Old GCs (real JVM)
Old GCs (simulation – last time)
Old GCs (simulation – latest)
Reference Counting or Tracing?

- Recall: Garbage collection algorithms
  - Tracing
    - What objects are reachable by the program?
  - Reference counting
    - Is there anybody holding a pointer to this object?

- Optimizations for reference counting

- Performance comparison with tracing
Tracing

- Reachability is determined by heap traversal in specific intervals
  - Out of memory, time interval, bytes allocated, ...
- Depth/breadth first search
  - starting from roots and advancing by scanning references from objects
Tracing

- Reachability is determined by heap traversal in specific intervals
  - Out of memory, time interval, bytes allocated, ...
- Depth/breadth first search
  - Starting from roots and advancing by scanning references from objects
Tracing

- Reachability is determined by heap traversal in specific intervals
  - Out of memory, time interval, bytes allocated, ...
- Depth/breadth first search
  - starting from roots and advancing by scanning references from objects
Tracing

- Reachability is determined by heap traversal in specific intervals
  - Out of memory, time interval, bytes allocated, ...
- Depth/breadth first search
  - starting from roots and advancing by scanning references from objects
Tracing

- Reachability is determined by heap traversal in specific intervals
  - Out of memory, time interval, bytes allocated, ...
- Depth/breadth first search
  - starting from roots and advancing by scanning references from objects
Reclaiming memory

- After establishing what objects are reachable
- Two basic approaches:
  - Traversal of all objects on the heap with deallocation of non-reachable objects
    - Mark & Sweep algorithm
  - Copy reachable objects to another memory area and destroy all objects left in original one
    - Semispace algorithm
Reference Counting

- Every object has an associated field that counts how many other objects do reference it.
- After an update of a reference, the count is updated.
Reference Counting

- Every object has an associated field that counts how many other objects do reference it.
- After an update of a reference, the count is updated.
Reference Counting

- Every object has an associated field that counts how many other objects reference it.
- After an update of a reference, the count is updated.
- Objects with no reference are deallocated.
Reference Counting

- Every object has an associated field that counts how many other objects reference it.
- After an update of a reference, the count is updated.
- Objects with no reference are deallocated.
  - If it has references, the counts of targets are updated.
Reference Counting

- Every object has an associated field that counts how many other objects reference it.
- After an update of a reference, the count is updated.
- Objects with no references are deallocated.
  - If it has references, the counts of targets are updated.
Reference Counting

• Every object has associated field that counts how many other objects do reference it

• After an update of a reference, the count is updated

• Objects with no reference are deallocated
  • If it has references, the counts of targets are updated
Reference Counting

- Every object has an associated field that counts how many other objects reference it.
- After an update of a reference, the count is updated.
- Objects with no references are deallocated.
  - If it has references, the counts of targets are updated.
Reference Counting

- Every object has an associated field that counts how many other objects do reference it.
- After an update of a reference, the count is updated.
- Objects with no reference are deallocated.
  - If it has references, the counts of targets are updated.
Reference Counting

- Problem:
  - Cycles
Reference Counting

- Problem:
- Cycles
Reference Counting

- Problem:
  - Cycles

Not reachable by the program
Reference Counting

- Problem:
  - Cycles

- Possible solutions:
  - Avoid cycles
  - Do nothing
  - Run tracing GC after some time
  - Trial deletion

Not reachable by the program
Advantages of Reference Counting

- Immediate reclamation
- Incremental
- Based on local information, not global processing

- Basic reference counting is easy to implement
  - Only a storage for counter and a write barrier is needed
  - Simplest tracing requires root enumeration
Limitations

- Cycles
- Simple implementation is also very slow
  - Need for interception of all pointer mutations
    - Heap
    - Variables
    - Stacks
    - Registers
- Smarter implementations avoid direct interception of stack and register pointer mutations
  - Still >30% slower than mark&sweep
Optimization Process

- Key design point identification
  - Evaluation of workload behavior in these points
    - How to store the counts
    - How to update the counts

- Application and evaluation of optimizations:
  - Reducing reference counter size
  - Reducing reference counter operations for new objects
  - Considering newly allocated objects as dead already
Storing the Count

• Adding a field
  • 32-bit word on a 32-bit system will be always enough
  • Adding this field causes overhead of 2.5% in total time and 6.2% in GC time (using tracing GC)

• Using only few bits that should be available in object header
  • Need to handle overflows
    – Do nothing
    – Additional data structure (hash table)
Maximum Reference Count Histogram

The graph shows the percentage of objects against the maximum reference count. Each line represents a different dataset, such as 'compress', 'db', 'mpegaudio', 'jack', 'bloat', 'eclipse', 'hsqldb', 'lusearch', 'pmd', 'xalan', 'jess', 'javac', 'mtrt', 'avrora', 'chart', 'fop', 'luindex', 'pjbb2005', 'sunflow', and 'mean'. The x-axis represents the maximum reference count, while the y-axis shows the percentage of objects. The legend on the right side of the graph indicates the color codes for each dataset.
Reference Count Storage Evaluation

- >99% of object reference counts can be stored within 3 bits
- However, the objects that overflow will cause ~23% of counter operations
  - The mechanism to handle overflows need to be very effective
- With limited counter size, 3 scenarios evaluated:
  - Hash table
  - Stuck counter + ignoring future count changes
  - Stuck counter + restoring counter in backup cycle tracing
Reference Count Storage Evaluation

(a) Total Time

(b) GC Time
Maintenance of Reference Counters

- Immediate reference counting
  - Simple, high overhead
- Deferred reference counting
  - Ignores pointer mutations in registers and on stacks
    - Zero count table + periodical root set enumeration
    - Objects in zero table and not in root set are garbage
- Coalescing
  - In a given window, only initial and final state of a reference are needed to determine correct reference counts
    - At mutation time, object is marked dirty and pointer values are stored
    - Subsequent operation on that (dirty) object are ignored
    - At collection, remembered objects are scanned, with increments on all outgoing pointers, decrements on stored pointers
Goal: Minimize the Number of Operations

- What objects cause most operations with reference counters?
  - New objects
    - Most of them die young
  - Objects with small maximum reference count

<table>
<thead>
<tr>
<th>Maximum count</th>
<th>Mean (opers., %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8-15</td>
<td>3</td>
</tr>
<tr>
<td>16-31</td>
<td>2</td>
</tr>
<tr>
<td>32-63</td>
<td>7</td>
</tr>
<tr>
<td>&gt;63</td>
<td>17</td>
</tr>
</tbody>
</table>
New Objects - RC Operations

(a) Increments

(b) Decrements
New Object Handling

- In deferred RC with coalescing:
  - New object is allocated *dirty*, RC = 1
  - Added to *decrement* buffer and *modified* buffer
  - At collection time
    - *Decrement* buffer is processed to compensate for initial setting (RC = 1)
    - *Modified* buffer is processed
      - Increment for each pointer target
Optimization: Lazy *Modified* insertion

- New objects are not added automatically into *modified* buffer
- Create 'new' bit in object header
- Increment processing
  - If target is 'new'
    - Clear 'new'
    - Add object to *modified* buffer

*Modified* processing causes cascade of increments
- Little bit of tracing masked as reference counting
Operations Reduction

(a) Increments

(b) Decrements
Optimization: New Objects Are Dead

- Most objects die young
- New objects allocated with RC = 0
- No decrement buffer to compensate
- Reference counter does not identify dead new objects, it identifies live new objects
  - Saves pointer scanning and decreasing RC in referenced objects
- Authors do not mention how the objects that need to be removed from memory are identified
New Objects Optimizations Evaluation

(a) Total time

(b) GC time
All Optimizations Together - Evaluation

(a) Total time

(b) GC time
Evaluation: RC vs Mark&Sweep

(a) Total time

(b) GC time
Conclusion

- Evaluation of bottlenecks in reference counting
- Improved performance of a contemporary reference counting collector
  - 24% faster in total time (average over tested workloads)
  - 74% faster in GC time
- Reference counting implementation can perform on par with well-tuned mark&sweep implementation
  - And only slightly worse than the best today's collectors
Thank you for your attention!