Garbage Collection: Overview and Modeling Experiments

Peter Libič
Department of Distributed and Dependable Systems

http://d3s.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
Outline

- Overview of GC algorithms
  - Recapitulation
  - Contemporary collectors present in the HotSpot VM

- Towards modeling
  - Proposed model
  - Issues
Part I: Garbage Collection Algorithms
Requirements

• Correctness
  • No deallocation of objects reachable by the program
    - Reachable from a set of roots
      • References from global/static objects/variables
      • References from objects/variables on stack

• Completeness
  • Every single piece of garbage will be eventually collected

• Performance

...
Collection Algorithms

- Two basic options:
  - Reference counting
  - Tracing
    - Mark & Sweep
    - Semispaces

- Basic optimization
  - Generational hypothesis
    - Objects that have been live for a long time are less likely to become garbage than objects that are “younger”
Reference Counting

- Every object has associated a 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 associated a 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 associated a 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.
Reference Counting

- Every object has associated a 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 associated a 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 associated a 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 associated a 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 associated a 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 from roots
Reference Counting

• Problem:
  • Cycles

• Possible solutions:
  • Avoid cycles
  • Run tracing GC after some time
  • Combination with tracing
    - Observation:
      • Cycles are present only if the reference count was decreased to a non-zero value

Not reachable from roots
**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
Mark & Sweep

- Two phases
  - Marking phase
    - At the heap traversal the objects that are proven reachable are flagged
  - Sweeping phase
    - All heap is traversed and allocated objects that do not have a flag are deleted
Mark & Sweep

● Two phases
  ● Marking phase
    – At the heap traversal the objects that are proven reachable are flagged
  ● Sweeping phase
    – All heap is traversed and allocated objects that do not have a flag are deleted
Mark & Sweep

- Two phases
  - Marking phase
    - At the heap traversal the objects that are proven reachable are flagged
  - Sweeping phase
    - All heap is traversed and allocated objects that do not have a flag are deleted

After Marking:
Mark & Sweep

• Two phases
  • Marking phase
    – At the heap traversal the objects that are proven reachable are flagged
  • Sweeping phase
    – All heap is traversed and allocated objects that do not have a flag are deleted

Sweep:
Mark & Sweep

- Two phases
  - Marking phase
    - At the heap traversal the objects that are proven reachable are flagged
  - Sweeping phase
    - All heap is traversed and allocated objects that do not have a flag are deleted

Sweep:
Mark & Sweep

• Two phases
  • Marking phase
    – At the heap traversal the objects that are proven reachable are flagged
  • Sweeping phase
    – All heap is traversed and allocated objects that do not have a flag are deleted

After sweep:
Semispace

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Semispace

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Semispace

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Semispace

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Semispaces

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Semispace

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Semispace

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Semispace

- Uses two equally sized memory areas
- All objects are allocated in one area
- Collection:
  - While the heap is traversed, each object the traversal encounters is copied into another area
  - In the first area there are no reachable objects
Advantages and Disadvantages

- Semispace is compacting
  - Simple and fast allocation ("bump-the-pointer")
- Semispace can use only half of total space
- Semispace needs to update references
- Mark & Sweep traverses the heap twice
- Both algorithms are using "stop-the-world" strategy
  - Application code is suspended for the time of collection
Generational collectors

• Observations:
  • Generational hypothesis
    – Older objects are less likely to become garbage than younger ones
    – Freshly allocated objects are more likely to become garbage than older ones
  • Complexity of tracing collectors depends (among other things) on number of live objects

• Try to collect young objects within smaller space more frequently
Generational collectors

- Split the heap into two (or more) generations
  - Usually, the higher the generation, the more space it uses
- Allocate new objects into the youngest generation
- If object survives certain number of young generation collections, promote it into higher generation

- Advantage: Bigger higher generations are collected less frequently
Generational collectors

X a = new X()
Generational collectors

Stack

Young GC

Generation is full => collection

Stack
Generational collectors

After several allocations, heap is full => collection

Full GC
Generational collectors - the catch

- References from higher to lower generation are problematic

- Workaround:
  - Instrumentation of reference modification by a write barrier
  - References are logged in remembered set
  - References from remembered set are added to the roots for lower generation collection
Write Barrier in HotSpot VM

- In HotSpot VM generational collectors the write barrier is implemented as follows:
  - **Card table**
    - Old generation heap space is partitioned in 512 bytes long chunks called *cards*.  
    - For every card there is 1 byte table entry with binary value:
      - There may be or there is no reference from memory represented by this field into young generation
  - **Write barrier**
    - At every reference update, it is checked if its source is in older generation and target in younger and if so, the card table entry is updated to “may be”
    - The generation membership can be determined by a simple comparison of addresses
  - At young generation collection the card table is scanned and references from corresponding cards that still exist are added to collection roots
Concurrent Mark & Sweep

- Collection runs parallel with the application code
  - except for very short pauses
- Concurrent marking
  - Snapshot at the beginning
    - Mark roots and direct successors of roots
  - Continue marking in parallel with application code
  - Every object allocated after snapshot is considered marked
  - Updated references are added to the list of visited but unprocessed objects (for BFS)
  - Short pause at the end to process remaining objects in the list
Concurrent Mark & Sweep

- Objects not marked after pause at the end of marking are garbage

- Sweep deallocates all objects that are not marked and were allocated before end of marking

- Concurrent version has lower throughput, but significantly shorter pause times
CMS: Example
CMS: Example - Initial Pause

Diagram:

- A
  - B
    - D
    - E
      - F
  - C
  - G
    - H
CMS: Example - Marking

Diagram of a tree structure with nodes labeled A, B, C, D, E, F, G, H. Node A is marked with an X.
CMS: Example - Marking

A → B
A ← C
C ← B

A, C, B, F
new J();

```
A  -->  B  -->  G
    |    /
    v    

C  -->  D  -->  E
    |    /    
    v    v    

F  -->  H  -->  J

A  -->  X  -->  C  -->  B  -->  F  -->  J
```
CMS: Example - Reference Modification

A → B → D → E → F → J

C.x = J
CMS: Example - Reference Modification
After several steps...

C.x = null
CMS: Example - At the End

Garbage and marked
G1 Collector

- Should be a default collector for server class machine of Sun HotSpot VM in version 1.7
- Authors claim it meets soft real-time goals with high probability while achieving high throughput
  - Based on heuristics
- Heap is partitioned into a set of equal-sized *regions*
- Every region has its remembered set of references that have source in a different region
- It uses “snapshot-at-the-beginning” concurrent marking algorithm
- For *evacuation* it selects several regions - *collection set*
G1 Collector

- Size of collection set is determined according to real-time goals
  - Defined as a portion of time within any time slice
  - For every region, it computes expected duration of the collection
  - Selects the set that should not exceed the goal
- Objects from collection set are copied (evacuated) into one or more empty regions
  - Only those that are not proven to be dead by concurrent marking
Part II: Garbage Collection Modeling?
Why do we want GC models?

- GC is integral part of many systems
- Sometimes imposing significant overhead
  - DaCapo benchmarks:
    - Eclipse workload - GC overhead:
      - ~3% with heap size ca 8 times the size of the footprint
      - 10 – 15% with heap size ca 4 times the size of the footprint
      - 20 – 30% with heap size ca 2 times the size of the footprint
  - GC behavior is not always 'linear'
  - If we want a model of a system, can we neglect such an important subsystem as the GC is?
What do we want from it?

- Collectors can be very complex
  - Significant simplifications required
- We want to assess:
  - How often is the collection started?
  - How long will it take?
Simple GC Model

- Premises:
  - Fixed size of the heap (denoted as HS)
  - Collections occur when the heap is full
  - Every collection is complete (no garbage left)
  - Objects are allocated one after another and they are numbered in a sequence starting from 1

- Input:
  - HS
  - Sequence $\text{SIZE}[i]$ of object sizes
    - Object with number $i$ has size $\text{SIZE}[i]$
  - Sequence $\text{DEATH}[i]$ of object death “times”
    - Object $i$ became unreachable after object $\text{DEATH}[i]$ was allocated and before object $\text{DEATH}[i] + 1$ was allocated
Simple GC Model

\[ HS = \left( \sum_{j=n_{i-1}+1}^{n_i} SIZE[j] \right) + \left( \sum_{j \in \{1...n_{i-1}\} \mid DEATH[j] \geq n_{i-1}} SIZE[j] \right) \]

- \( n_i \) – collection number \( i \) occurred after allocating \( n_i \) objects
Simple GC Model

\[ HS = \left( \sum_{j=n_{i-1}+1}^{n_i} \text{SIZE}[j] \right) + \left( \sum_{j \in \{1...n_{i-1}\} \atop \text{DEATH}[j] \geq n_{i-1}} \text{SIZE}[j] \right) \]

- \( n_i \) — collection number \( i \) occurred after allocating \( n_i \) objects

Size of the objects allocated since last collection

Size of the objects that survived last collection
Simple GC Model

\[ HS = \left( \sum_{j=n_{i-1}+1} \text{SIZE}[j] \right) + \left( \sum_{j \in \{1\ldots n_{i-1}\}, DEATH[j] \geq n_{i-1}} \text{SIZE}[j] \right) \]

- \( n_i \) – collection number \( i \) occurred after allocating \( n_i \) objects

Variable

Size of the objects allocated since last collection

Size of the objects that survived last collection
Simple GC Model

- Approximate solution for:
  - Average object size – OS
  - Average object lifetime – LT
    - DEATH[i] = LIFETIME[i] + i

\[ \Delta n = \frac{HS}{OS} - LT \]
Generational GC Model

- Premises:
  - Young and old generation
  - Fixed size of both generations heap space
    - denoted as YG for young and OG for old generation
  - New objects are allocated into young generation
  - Young generation collections occur when it is full
  - Old generation collection occurs together with some of young generation collections, when the old generation is full
  - Every collection of a generation is complete (no garbage left)
  - Objects from young generation are promoted to old generation after surviving PCC young generation collections
  - Objects are allocated one after another and they are numbered in a sequence starting from 1
Generational GC Model

- **Input:**
  - YG, OG, PCC, SIZE[i], DEATH[i]

\[
YG = \left( \sum_{j=n_{i-1}+1}^{n_i} \text{SIZE}[j] \right) + \left( \sum_{j \in \{n_{i-1-PCC} \ldots n_{i-1}\} \atop \text{DEATH}[j] \geq n_{i-1}} \text{SIZE}[j] \right)
\]

- Size of objects allocated since last collection
- Size of objects that survived last young generation collection and were not promoted to the old generation – objects will not be in young generation for more than PCC collections
Generational GC Model

\[ OG = \left( \sum_{j: \exists m_i > m_{i-1}} \text{SIZE}[j] \right) + \left( \sum_{j: \exists l : n_l = m_{i-1}} \text{SIZE}[j] \right) \]

- \( m_i \) – full collection number \( i \) occurred after allocating \( m_i \) objects
- Or – full collection occurred after \( k-l \) young generation collections
Generational GC Model

\[ OG = \left( \sum_{j:} SIZE[j] \right) + \left( \sum_{j:} SIZE[j] \right) \]

- Size of objects promoted into old generation since last full collection
- Size of objects that survived last old generation collection

**Example: PCC = 3**

Objects allocated before this point were already promoted at \( n_l \) collection

Objects allocated here could be promoted in between last two collections

Objects allocated after this point could not be promoted before \( n_k \) collection

Young collections - \( n_i \)

Full colls. - \( m_i \)

\( k-l = 2 \)
Generational GC Model

- Approximate solution for:
  - Average object size – OS
  - Average object lifetime – LT
- Condition \( n_{i-1-PCC} + 1 \geq n_{i-1} - LT \)
  - Should be valid for reasonable workloads

\[ \Delta n = \frac{YG}{OS \cdot (1 + PCC)} \]

\[ k - l = \frac{OG}{OS} + 2 \cdot PCC \cdot \Delta n - 2 \cdot LT \]

\[ k - l = \frac{OG}{OS} + 2 \cdot PCC \cdot \Delta n - 2 \cdot LT \]

\[ \Delta n \]
First Experiments

- Sunflow benchmark from DaCapo suite
- Parameters measured with modified Jikes RVM machine
- Measured values and configuration:
  - LT = 230000, OS = 34B, HS = 32M
  - Predicted $\Delta n = 757000$ (Simple model)
  - Real $\Delta n = 610000$
- Every heap implementation has some overhead – at least alignment and storage of object length
  - Let's guess we have 8B alignment $\rightarrow$ OS = 40B :-)
  - Predicted $\Delta n = 608000$ (Simple model)
- Generational model does not work at all :-(

Department of Distributed and Dependable Systems
What's next?

- Find out why the generational model does not work and fix it
- Find a better solution for model equations
  - Probability distribution as a parameter
Thank you for your attention!

Questions?