Generation Scavenging
A Non-Disruptive High Performance Storage Algorithm

David Ungar
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Presented By:
Sundeep Kushwaha
Organization Of Presentation

- Introduction
- Relationship: Virtual Memory And Storage Reclamation
- Bandwidth Issues With Storage Allocator
- Various Garbage Collection Algorithms
  - Reference Counting (RC)
    - Immediate RC
    - Deferred RC
  - Marking Storage Reclamation Algorithms
    - Mark and Sweep
    - Scavenging
- Overview Generation Scavenging Algorithm (GSA)
- Comparison of GSA with other Scavenging Algorithms
- Evaluation of GSA
- Conclusion
- Questions / Discussion
Word Of Caution

- This presentation is going to be my interpretation of Generation Scavenging.
- This paper was published in 1984. To appreciate ideas presented in this paper we should read it with right mind set.
Introduction To Generation Scavenging Algorithm

- Computing systems provide automatic storage facilities
- Price to be paid:
  - CPU Time
  - Main Memory
  - Unexpected pauses cause distraction and reduction of productivity
- Proposed Generation Scavenging Algorithm (GSA)
  - Limits pause times to a fraction of a second
  - Requires no hardware support
  - Meshes well with virtual memory
  - Reclaims circular structures, and
  - Uses less than 2% of CPU time on Smalltalk system
- GSA has been implemented on Berkeley Smalltalk (BS)
Relationship: Virtual Memory and Storage Reclamation

- CPU
- Virtual Memory Address Space
- Main Memory
- Secondary Storage
- 4GB VAS
- 32-bit Address
- Address Translation
- Paging
Bandwidth Issues With Storage Allocator

- Bandwidth is the reclamation rate for system to be in equilibrium.
- Smalltalk-80 system allocates a new object every 80 instructions.
- Mean dynamic object size is about 70 bytes.
- If system runs at 9000 bytecodes per second:
  - Storage Allocator Bandwidth =

  \[
  \text{70 bytes/1 object } \times \text{1 object/80 instructions } \times \text{9000 bytecodes/second} = 7800 \text{ b/s}
  \]

- What does this mean?
Bandwidth Issues With Storage Allocator

- Flush out data from main memory to secondary storage at 7800b/s

\[ T = \frac{100 \text{ MB}}{7800 \text{ BpS}} = 3.5 \text{ Hrs} \]

- Recycle data from Main Memory (GC)
Various Garbage Collection Algorithms

- Reference Counting (1960):
  - Maintain a count of number of pointers that reference each object
    - Immediate RC:
      - Adjust reference count on every store instruction
      - Counting references takes time. Around 15% of CPU time
      - Additional 5% for decrementing counts when object is released
      - Advantages: least amount of memory for dynamic objects
      - Fails to reclaim circular structure
    - Deferred RC:
      - Ignore references from local variables
      - Preclude reclamation during program execution
      - System has to periodically stop to free dead objects
      - Requires 25 KB more space as compared to Immediate RC
      - 30 ms pause every 500 ms
      - Saves 90% of reference count manipulation
      - 3% CPU Time + 3% periodic reconciliation + 5% for recursive freeing
Various Garbage Collection Algorithms

- Marking Storage Reclamation Algorithms (1960):
  First traverse and mark reachable objects and then reclaim the space filled by unmarked ones
  - Mark and Sweep
    - Marking phase identifies all live objects
    - Reclaims one object at a time.
    - Inefficient, because this algorithm requires object space to be traversed twice.
    - CPU Time: 25%-40%
    - 4.5 second pause every 79 seconds
  - Scavenging Live Objects
    - Costly sweep phase can be eliminated by moving live objects to a new area
    - After scavenging former area is free and new objects can be allocated from its base
    - Forwarding pointers are required
    - CPU Time: 7%
    - Next improvement is to divide objects into generations and do GC more often for younger ones.
Generation Scavenging Algorithm

- Each object is classified as new or old
- Old objects reside in memory region called old area
- New objects can be found in following places
  - NewSpace
  - PastSurvivorSpace
  - FutureSurvivorSpace
- Remembered Set: Set of old objects having a reference to new object
- All new objects are reachable through Remembered Set objects and roots
- During GC, live objects from NewSpace and PastSurvivorSpace are moved to FutureSurvivorSpace
- Interchange FutureSurvivorSpace with PastSurvivorSpace
- NewSpace can be reused for new objects
- Space cost of only 1 bit/object
- Tenuring: promotion from new space to old space
Generation Scavenging Algorithm

Old Object Space

Rem Set

New Object Space

NewSpace
140 KB

PastSurvivorSpace
28 KB

FutureSurvivorSpace
28 KB

Registers
Generation Scavenging Algorithm
Generation Scavenging Algorithm

- Registers
- Old Object Space
- Rem Set
- New Object Space
- NewSpace
- PastSurvivorSpace
- FutureSurvivorSpace
- New Object Space
Generation Scavenging Algorithm: Tenuring

- Registers
- Old Object Space
- Rem Set
- New Object Space
- FutureSurvivorSpace
- PastSurvivorSpace
- NewSpace
GSA: Role Of Virtual Memory

CPU → Virtual Space → Main Memory (NS, FSS, PSS, OS2, RS1) → Secondary Storage (OS2, RS2) → Paging

Main Memory:
- NS
- PSS
- FSS
- RS1
- OS2

Secondary Storage:
- OS2
- RS2
Comparison of GSA with other scavenging algorithms

- **Similarities**
  - It divides objects into young and old generations
  - Copies live objects instead of sweeping dead ones
  - Reorganizes old objects offline

- **Differs**
  - Conserves memory space by dividing new space into three spaces instead of two
  - Is not incremental. This eliminates the checking needed for load instructions
Evaluation of GSA

- CPU Time:
  - Takes only 1.5% of total user CPU Time
  - This is four times better than its nearest competitor (7%)

- Main Memory Consumption:
  - Takes only 200 KB (140 + 28 + 28) for dynamic objects
  - Around 10% of BS main memory
  - Comparison with Baker Semispace Algorithm: 2 * (140+28) = 360 KB (apx)

- Pauses
  - Pauses were small averaging 150 ms
  - Longest was 330 ms
Conclusion

- Combination of generation scavenging and paging provides high performance GC
- Careful consideration of virtual memory is essential for any GC algorithm
- GSA uses these principles to achieve 2% CPU time, 200 KB primary memory, 1.2/s backing store operations and 1/6-1/3 s pause time.
Discussion

- Do we have a control over paging?
- Is it still a good idea to page out old object space to secondary memory?
- Are the results reliable? He used only (I guess) smalltalk-80 macro benchmarks.