Garbage Collection Techniques

Michael Jantz
Memory Management

• Memory Management
  – Recognizing when allocated objects are no longer needed
  – Deallocating (freeing) the memory used by such objects

• Explicit in some languages (C, C++, Fortran, Pascal)
  – Easy to understand what is going on
  – Can be more efficient when there is a shortage of memory

• Drawbacks of explicit memory management
  – Programmer has to do it (extra code)
  – Memory management bugs (dangling pointers, memory leaks, etc)
Automatic Memory Management
(or Garbage Collection)

• Garbage collector is responsible for:
  – Allocating objects in memory
  – Automatically recovering memory used by objects that are no longer reachable

• Advantages of GC
  – Programmer does not have to manage memory!
  – Fewer memory management bugs
  – Increased abstraction of interfaces / more reliable code (no pointers)

• Drawbacks of GC
  – Memory may be retained for too long (even if it will not be used again)
  – Can require significant space / performance overhead
  – Not available with every language / platform
Recommended Reading

• This presentation incorporates material from *Uniprocessor Garbage Collection Techniques* by Paul R. Wilson
  – Well-cited survey written in 1992
  – Many techniques discussed by Wilson still in use today
Phases of Garbage Collection

• Two phases of GC
  – Garbage detection
  – Reclamation of garbage objects' storage

• Phases can be functionally or temporally interleaved – and the reclamation phase strongly depends on the detection phase
Garbage Detection

• Object *liveness* defined in terms of *reachability* from a *root set*
  – *Root set* includes objects known to be live at time of GC (global variables, local variables of active procedures)
  – Any objects directly reachable from the root set are live
  – Any object reachable from any live object is also live
  – Unreachable objects are *garbage*

• Live objects must be preserved

• Garbage objects can be safely reclaimed
Object Representation

• Assume heap objects are self-identifying
  – Most statically-typed languages include a "header" field on heap objects
  – The header contains type information that can be used to decode the format of the object
Basic Garbage Collection Techniques

• Reference Counting
• Mark-Sweep
• Mark-Compact
• Copying Collector
• Non-copying Implicit Collector (Baker's Algorithm)
Reference Counting

• Each object maintains associated count of references (pointers) to it
  – Each time a reference to an object is created, increment the counter
  – When an existing reference is eliminated, decrement the counter

• When the counter reaches zero, reclaim the object

• When an object is reclaimed, any the counters for any objects it points to are also decremented
  – Potentially leading to a cascading effect and reclaiming of many other objects
Reference Counting

Figure from Wilson, 1992
Reference Counting

• Advantages
  – Collection work interleaved closely with program execution (can easily be made incremental and real-time)
  – No extra work if number of live objects is large
  – Immediacy of reclamation can improve reference locality

• Disadvantages
  – Not always effective (cannot remove cyclic garbage)
  – Difficult to make efficient
Reference Counting Effectiveness

Figure from Wilson, 1992
Reference Counting Effectiveness

Figure from Wilson, 1992
Reference Counting Efficiency

• Cost is typically proportional to amount of work done by the program
• Extra costs
  – Adjust a pointer's reference counts when it is created or destroyed
  – Changing a variable from one pointer to another requires two updates
    (increment and a decrement)
  – Objects passed on the stack require an increment and decrement when the
    function is called and when it returns
Reference Counting Optimizations

• Deferred reference counting
  – Do not include bookkeeping for references from local variables (on the stack) most of the time
  – Always adjust reference counts for pointers from one heap object to another
  – Periodically scan the stack and reclaim garbage after taking into account stack objects

• Smart pointers (C++)
  – Passing pointers by reference avoids reference count adjustments on function enter and exit

• Remove redundant updates in between reclaim intervals
Mark-Sweep Collection

• Detection (mark) phase
  – Starting from the root set, traverse the graph of pointer relationships.
  – Mark objects that are reached in some way

• Reclaim (sweep) phase
  – After marking live objects, sweep memory to find all the unmarked (garbage) objects and reclaim their memory
Mark-Sweep Collection Example

Example from slides by Vitaly Shmakinov, Fall 2010, UT-Austin
Mark-Sweep Collection Example

Example from slides by Vitaly Shmakitov, Fall 2010, UT-Austin
Mark-Sweep Collection Example

Example from slides by Vitaly Shmakitov, Fall 2010, UT-Austin
Mark-Sweep Collection Example

Example from slides by Vitaly Shmakitov, Fall 2010, UT-Austin
Mark-Sweep Collection

• The good:
  – Handles cycles correctly
  – No space overhead (mark bit fits in space allocated for pointer)

• The bad
  – Normal execution has to be suspended
  – Cost of a collection is proportional to the size of the heap
  – Heap fragmentation
  – Hurts locality of reference
Mark-Compact Collection

• Solution for fragmentation problem in Mark-Sweep
• Same as Mark-Sweep, except during sweep phase
  – Compact live objects to front of heap space until all lives are contiguous
  – Remaining space is one contiguous free space
• Employs a linear scan through memory, and "slides" live objects down to the previous object
Mark-Compact Collection

- Heap before GC
- Green is header, Blue is object, Orange is free heap space

Example from slides by Gregor Richards, Waterloo
Mark-Compact Collection

• Marking phase identifies dark blue objects as still alive
• Lighter objects unmarked (dead)

Example from slides by Gregor Richards, Waterloo
Mark-Compact Collection

- Space is free, but live objects need to be compacted

Example from slides by Gregor Richards, Waterloo
Mark-Compact Collection

- Each live object is moved down to the adjacent previous object – filling all the memory holes.

Example from slides by Gregor Richards, Waterloo
Mark-Compact Collection

• End result is a single, large, contiguous area of free space at the end of the heap

Example from slides by Gregor Richards, Waterloo
Mark Compact Collection

• Advantages:
  – Solves the fragmentation issue
  – Locality is also improved

• Drawbacks
  – Still have to suspend the application
  – Requires several passes over the live data objects
    • 1\textsuperscript{st} pass: marking phase
    • 2\textsuperscript{nd} pass: compute new locations that live objects will be moved to
    • 3\textsuperscript{rd} pass: update all pointers to refer to new object locations
    • 4\textsuperscript{th} pass: actually move the objects
Copy Collection

• Move all live objects into a separate heap area
• Rest of heap is known to be garbage (and can be used for allocation)
• Similar, but not the same as mark-sweep and mark-compact
  – Traverses objects from root set to find live objects
  – Copying to a separate heap space is integrated with the traversal
• Sometimes called "scavenge" collectors
Simple Semi-Space Collector

• Space in heap divided into two contiguous *semispaces*
• Only one space is in use during normal operation
  – Allocation from the "current" space is simple and similar to mark-compact
  – No fragmentation problem
• When the program demands an allocation that is too large for the "current" space, program is stopped for garbage collection
Simple Semi-Space Collector

Figure from Wilson, 1992
Simple Semi-Space Collector

Figure from Wilson, 1992
The Cheney Algorithm (1970)

• All immediately reachable objects are placed into tospace
• The tospace maintains a "scan" and "free" pointer
  – The "scan" pointer advanced through the object in the tospace, location by location, to perform a BFS traversal of the fromspace
  – The "free" pointer marks the end of the tospace and is updated when live objects are copied over
• On encountering a pointer that points into the fromspace
  – fromspace object is copied into the tospace
  – The pointer is updated to point to new object location in the tospace
  – The "scan" pointer is then advanced and the scan continues
Figures from Wilson, 1992
The Cheney Algorithm (1970)

• What if an object is reached by multiple paths?
  – e.g. B → E instead of B → D on previous slide

• When an object is copied, install a *forwarding* pointer on the old version of the object
  – The forwarding pointer indicates where to find the new copy
  – Ensures all objects are copied exactly once and all pointers are updated to refer to the new object
Copy Collection Efficiency

• Effort in copying GC
  – Work done at each collection is proportional to amount of live data
  – If amount of live data is about the same at each collection, decreasing GC frequency will decrease GC effort

• How to decrease GC frequency?
  – Increase heap space!
  – If semispaces are larger, program will run longer before filling them up

• Example:
  – Program allocates 20MB of memory over the whole run, but only about 1MB live at any given time
Copy Collection Efficiency

- Assume semi-spaces are 3MB each
- GC occurs about 10 times (2MB free after each GC)

Figure from Wilson, 1992
Copy Collection Efficiency

- If size of semi-spaces is doubled to 6MB each
- GC only occurs about 4 times (5MB free after each GC)

Figure from Wilson, 1992
Non-Copying Implicit Collection
Baker's Algorithm

• Inspired by copying collector
  – GC spaces are a special case of sets
  – Another implementation of sets could do just as well (provided it has similar performance characteristics)

• Add two pointer fields and a "color" to each objects
  – Objects linked in a doubly-linked list by the pointer fields
  – Color indicates which set the object belongs to

• Allocate objects from a list of free-space objects

• When free-list exhausted, traverse the live object and "move" them from the 'from-set' to the 'to-set'
  – Unlink object from the 'from-set' list and insert into the 'to-set' list
  – Toggle the object's color field
Non-Copying Implicit Collection

Baker's Algorithm

• Observe that
  – Space reclamation is implicit (just like the original copy collector)
  – Objects do not have to actually be copied (after GC, 'from-set' is a list of free-space objects)

• The bad
  – Slightly higher space overhead per-object
  – Fragmentation is an issue again

• The good
  – Do not have to copy large objects
  – Do not even have to scan objects that are known not to contain pointers
  – Does not require language-level pointers to be changed
Incremental Tracing Collectors

• 'Stop-the-world' garbage collectors can be disruptive
  – With large heap, program becomes non-responsive for long periods of time
  – Unacceptable for real-time applications

• Reference counting works incrementally
  – But is less efficient and less effective than trace collectors

• Mutators are a problem for tracing collectors
  – During garbage detection, the "mutator" threads / procedures can change the graph of reachable objects

• An incremental scheme must track changes to reachable objects
  – How conservative should the GC be wrt changes made by the mutator?
Tricolor Marking

• Conventional tracing collectors use binary color markings
  – Objects subject to GC are white, objects known to be live are black
  – Trace completes when there are no more reachable nodes left to blacken

• Need some way of ensuring consistency between mutator and incremental collector

• Solution: introduce a third color: gray
  – Objects are colored gray if they have been reached by the traversal, but their descendants might not have been
  – Only blacken objects after pointers to their offspring are traversed
Tricolor Marking

• Important property: no pointers from black $\rightarrow$ white
  – Useful to assume that the collector is "finished with" the black objects
Incremental Approaches

• Read Barriers
  – If the mutator accesses a pointer to a white object, immediately color the white object gray

• Write Barriers
  – Trap / record when the mutator attempts to write a pointer into an object
  – Aim to address when the mutator:
    • Writes a pointer to a white object into a black object
    • Destroys the path to a white object before the collector sees it
Write Barrier Approaches

- **Snapshot-at-beginning**
  - Ensures that pointers to white objects are not destroyed
  - Saves all pointers at beginning of trace in case a path to a white object is over-written

- **Incremental update**
  - Record pointers stored into black objects
  - Black object is reverted to gray when the mutator "undoes" the collector's traversal
Baker's Incremental Copying Collector (1978)

• Real-time garbage collector
  – Adaptation of simple copy collector
  – Employs a read barrier to coordinate with the mutator

• Incremental collection
  – *Background scavenging* is interleaved with the mutator
  – New objects allocated to the 'to-space' (assumed to be live)
  – Rate of collection tied to the rate of allocation
  – Mutator coordination
    • When the mutator reads a pointer from the heap: first check, is it in the 'from-space'?; if yes, copy the referent to the 'to-space' (switch color from white to gray)
The Treadmill

• Non-copying version of Baker's incremental collector
• Links various lists into a cyclic structure with four sections
  – New: for new objects created during GC
  – From: holds objects allocated before GC began
  – To: for objects that survive GC (copied from the 'from-list')
  – Free: free list for new allocations
• After GC:
  – 'from-list' objects are known to be garbage and merged with the 'free-list'
  – 'to-list' and 'new-list' objects are known to be live (black) and are merged
The Treadmill
Generational Garbage Collection

• Most objects do not live for very long
  – According to Wilson, 80% to 98% die within a few million instructions

• A large portion of those that survive at least one GC will survive through many collections
  – These objects are copied at every GC
  – Results in a lot of wasted effort

• Generational Collection
  – Separate young objects from old objects
  – Scavenge areas with old objects less often than young objects
  – Once objects have survived some number of collections, move them to an area that is collected much less frequently
Multi-Generational Copying Collector

• Memory is divided into multiple areas (generations) that hold objects of different approximate ages
• Each generation further divided into semi-spaces for copy collection
• If an object survives enough collections, it is copied from the young space to the old space (rather than back to the other semi-space)
• Eventually older generation will have to be collected as well
  – But much less frequently than the new generation
Multi-Generational Copying Collector
Multi-Generational Copying Collector
Multi-Generational Copying Collector

First (New) Generation Memory

Second Generation Memory
Detecting Intergenerational References

• Generational scheme requires you to scavenge young space without scanning all the older objects

• Need to detect pointers from old to new memory

• Potential solutions
  – Indirection tables
  – Pointer recording
  – Card Marking
Card Marking

• Heap divided into a set of cards
  – Each card is typically smaller than a page of memory

• Runtime maintains a card map
  – One bit for each card in the heap

• Whenever a pointer field for an object is modified, set the corresponding bit in the card map

• At GC time, dirty cards are scanned for objects containing references into the younger generation

Image from Alexey Ragozin
Performance of Garbage Collection

• Hertz and Berger study on the performance of automatic memory management vs. explicit memory management (OOPSLA 2005)
  – Employ an oracular memory manager execute unaltered Java programs as if they used explicit memory management
  – Examine performance (run-time), space consumption, and virtual memory footprint of Java programs across a range of allocators and GC's
Explicit Memory Management for Java

• Straightforward to measure the performance effect of GC in languages designed for explicit memory management
  – In C and C++, disable explicit frees and employ conservative (i.e. non-relocating) GC schemes

• How to measure cost of explicit memory management in a language designed for automatic memory management?
  – Cannot replace GC with explicit because programs never deallocate data
  – Cannot extrapolate from previous studies because re-locating GC's consistently outperform conservative schemes
Conventional Wisdom

• Widespread belief that explicit memory mgmt. performs better than automatic mgmt. due to cache locality and run-time overheads
  – “There just aren’t all that many worse ways to [expletive deleted] up your cache behaviour than by using lots of allocations and lazy GC to manage your memory” ... “GC sucks donkey brains through a straw from a performance standpoint” – Linus Torvalds (creator of Linux)
  – Garbage collection will be beaten "by a manual tracking system’s best case performance" – Dan Sugalski (architect of Perl 6)
  – "[garbage collection] will likely require more CPU time than would have been required if the program explicitly freed unnecessary memory” – Bill Venners, author of *Inside the Java Virtual Machine*
Oracular Memory Management

Step 1: Data Collection and Derivation of Death Records

- Profile run for calculating object lifetimes and generating the program heap trace
- Calculate reachability times for program objects offline (Merlin analysis)
• Using the oracle, simulate program execution
  • Object allocation replaced by calls to malloc
  • Objects are freed when directed by the oracle
Object Lifetimes

• Two types of oracle
  – Lifetime-based oracle: frees objects after their last use (earliest safe point)
  – Reachability-based oracle: frees objects immediately after they become unreachable (last point an explicit manager could free them)
Experimental Methodology:

Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Alloc</th>
<th>Max Reach</th>
<th>Alloc/Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>125,334,848</td>
<td>13,682,720</td>
<td>9.16</td>
</tr>
<tr>
<td>_202_jess</td>
<td>313,221,144</td>
<td>8,695,360</td>
<td>36.02</td>
</tr>
<tr>
<td>_205_raytrace</td>
<td>151,529,148</td>
<td>10,631,656</td>
<td>14.25</td>
</tr>
<tr>
<td>_209_db</td>
<td>92,545,592</td>
<td>15,889,492</td>
<td>5.82</td>
</tr>
<tr>
<td>_213_javac</td>
<td>261,659,784</td>
<td>16,085,920</td>
<td>16.27</td>
</tr>
<tr>
<td>_228_jack</td>
<td>351,633,288</td>
<td>8,873,460</td>
<td>39.63</td>
</tr>
<tr>
<td>ipsixql</td>
<td>214,494,468</td>
<td>8,996,136</td>
<td>23.84</td>
</tr>
<tr>
<td>pseudoJBB</td>
<td>277,407,804</td>
<td>32,831,740</td>
<td>8.45</td>
</tr>
</tbody>
</table>

Table 1: Memory usage statistics for our benchmark suite. Total allocation and maximum reachable are given in bytes. Alloc/max denotes the ratio of total allocation to maximum reachable, and is a measure of allocation-intensiveness.
Experimental Methodology
Measurement and Allocators

• Heap size range from smallest to 4x the smallest possible
• Measure *heap footprints*
  – Actual number of pages in use (allocated and touched)
• Conduct study with two allocators
  – Lea: allocator used by GNU C library
  – MSExplicit: modified Treadmill collector that includes free functionality
    • Each block maintains its own stack of free slots
    • Reuses the slot that has been most recently freed
Experimental Methodology:
Garbage Collectors

<table>
<thead>
<tr>
<th>Garbage collectors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MarkSweep</td>
<td>non-relocating, non-copying single-generation</td>
</tr>
<tr>
<td>GenCopy</td>
<td>two generations with copying mature space</td>
</tr>
<tr>
<td>SemiSpace</td>
<td>two-space single-generation</td>
</tr>
<tr>
<td>GenMS</td>
<td>two generations with non-copying mature space</td>
</tr>
<tr>
<td>CopyMS</td>
<td>nursery with whole-heap collection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allocators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lea</td>
<td>combined quicklists and approximate best-fit</td>
</tr>
<tr>
<td>MSExplicit</td>
<td>MMTk’s MarkSweep with explicit freeing</td>
</tr>
</tbody>
</table>

Table 2: Memory managers examined in this paper. Section 4 presents a more detailed description of the allocators and collectors.
Performance vs. Heap Footprint

- X-axis shows geo mean of relative heap footprint for 8 benchmarks
- Y-axis shows performance (simulator cycles)
- All results relative to Lea allocator with reachability oracle
Performance vs. Heap Footprint

- With small heap, cost of GC dominates run-time
- As heap sizes grow, execution time approaches a fixed value
Performance vs. Heap Footprint

- GenMS is most competitive with Lea
- MarkSweep does best with low-allocation intensity workloads
Comparison with GenMS

<table>
<thead>
<tr>
<th>Heap size</th>
<th>Footprint</th>
<th>Runtime</th>
<th>Footprint</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>210%</td>
<td>169%</td>
<td>253%</td>
<td>167%</td>
</tr>
<tr>
<td>1.25</td>
<td>252%</td>
<td>130%</td>
<td>304%</td>
<td>128%</td>
</tr>
<tr>
<td>1.50</td>
<td>288%</td>
<td>117%</td>
<td>347%</td>
<td>115%</td>
</tr>
<tr>
<td>1.75</td>
<td>347%</td>
<td>110%</td>
<td>417%</td>
<td>109%</td>
</tr>
<tr>
<td>2.00</td>
<td>361%</td>
<td>108%</td>
<td>435%</td>
<td>106%</td>
</tr>
<tr>
<td>2.25</td>
<td>406%</td>
<td>106%</td>
<td>488%</td>
<td>104%</td>
</tr>
<tr>
<td>2.50</td>
<td>419%</td>
<td>104%</td>
<td>505%</td>
<td>102%</td>
</tr>
<tr>
<td>2.75</td>
<td>461%</td>
<td>103%</td>
<td>554%</td>
<td>102%</td>
</tr>
<tr>
<td>3.00</td>
<td>476%</td>
<td>102%</td>
<td>573%</td>
<td>100%</td>
</tr>
<tr>
<td>3.25</td>
<td>498%</td>
<td>101%</td>
<td>600%</td>
<td>100%</td>
</tr>
<tr>
<td>3.50</td>
<td>509%</td>
<td>100%</td>
<td>612%</td>
<td>99%</td>
</tr>
<tr>
<td>3.75</td>
<td>537%</td>
<td>101%</td>
<td>646%</td>
<td>100%</td>
</tr>
<tr>
<td>4.00</td>
<td>555%</td>
<td>100%</td>
<td>668%</td>
<td>99%</td>
</tr>
</tbody>
</table>

Table 4: Geometric mean of memory footprints and runtimes for GenMS versus Lea. The heap sizes are multiples of the minimum amount required to run with GenMS.

- Performance of GC tends to be inversely proportional to heap size
- With explicit MM, performance does not depend on heap size
Page-Level Locality

- LRU scheme for paging when memory size is fixed
- Assume 5ms penalty for page faults
Page-Level Locality

- x-axis: amount of available memory (MB)
- Y-axis: execution time (in seconds, log scale)
Page-Level Locality

- With fixed amount of memory, explicit memory managers outperform all GC's.
- GC activity visits far more pages than the application, and kills page locality.
Conclusions: Automatic vs Explicit MM

• Methodology executes unaltered Java programs as if they used explicit memory management

• Execution time of best-performing GC is competitive with explicit MM when given enough memory

• GC performance degrades when forced to use smaller heaps
  – GC is more susceptible to paging when physical memory is scarce

• Recommendation:
  – If you have (at least 3x) more memory than you actually need, GC is great!
  – If your application will be memory constrained, GC will hurt performance
Backup
Oracle Comparison

![Graph: Geometric Mean of Simulated Cycles by Relative Number of Heap Pages]

- Lea w/Reachability
- Lea w/Liveness
- MSExplicit w/Reachability
- MSExplicit w/Liveness

Geometric Mean of Relative Simulated Cycles
Geometric Mean of Relative Heap Footprint
MSExplicit vs Lea Allocator

- MSExplicit allocator not as space efficient as Lea
- Performs better with liveness oracle