Garbage Collection Survey

- Based on Wilson's short survey

Garbage Collectors

- Two different tasks:
  - reachability computation (from *root set*)
  - reclamation
- Main degrees of variability:
  - is GC incremental or stop-and-collect?
    - incremental better, more complex
  - are objects moved during reclamation?
    - if not, high complexity to link garbage in free lists, etc.
    - if they are, we lose conservatism (have to know all pointers), may suffer from the copy overhead (usually not too important)
- Important question: How to tell pointers from data?
  - type information
  - header fields, tag bits, etc.

Basic Garbage Collection Techniques

1. Reference Counting collection
2. Mark-Sweep collection
3. Mark-Compact collection
4. Copying collection
5. Implicit, non-copying collection

1. Reference Counting

- Keep for each object a count of how many times it is pointed to
- Advantages:
  - simplicity, easy to make incremental
  - locality probably good [DeTreville]
  - little degradation when heap is almost full
  - objects stay in place, can be conservative
- Disadvantages:
  - work done for each pointer copying
    - horrible for short-lived variables
  - cycles can fool it
    - cycles are common (doubly linked lists, etc.)
    - biggest showstopper

2. Mark-Sweep Collection

- Two phases: *marking* (for reachability computation), *sweeping* (for reclamation)
- Advantages:
  - objects stay in place, can be conservative
- Disadvantages:
  - allocator kind of problems: fragmentation, locality
  - cost of collection proportional to amount of garbage + live data (not just live data)
- All these problems can be alleviated with clever techniques, as in allocators. On the other hand, they can be solved altogether with other GC techniques

3. Mark-Compact Collection

- Like mark-sweep, only compaction phase instead of sweeping (usually "sliding compaction")
- Advantages:
  - no fragmentation
  - good locality after compaction
- Disadvantages:
  - no conservatism
  - horrible locality during GC (multiple passes through memory)

4. Copying Collection

- Addresses a problem of mark-compact: memory needs to be traversed once
  - reachability computation and copying are interleaved
  - *scavenging* is another term used
- Most common implementations: semi-spaces with "scan" and "free" pointers that implement the tricolor abstraction:
  - copied and scanned objects are black
  - copied but not scanned objects are grey
  - not copied, not scanned objects are white
- A FIFO scan queue (implementing a breadth-first traversal) is often used for copied blocks
- Forwarding pointers are needed for objects that get copied (the originals may be reachable through some other objects—these references should be updated)

How often to copy (or compact)?

- " As rarely as possible, but definitely not more rarely"
- Infrequent collections are great: objects die
- But heap cannot be too large: has to fit in memory for collector to avoid paging
- Disadvantage of copying (relative to compact): semi-spaces make matters worse

5. Implicit Collection

- Observation: merging list is very fast
- Keep all objects in lists. Distinguish three lists:
  - allocated
  - free
  - live (only used during GC)
- Implicit non-copying collection works isomorphically to a common copying collector:
  - scan to “move” objects to live list (not really moved)
  - then merge the allocated and free lists (allocated contains only garbage)
- Advantages:
  - no copying (conservative, etc.)

- Disadvantages:
  - no copying (fragmentation, etc.)
  - high overhead (even allocated blocks need to have list pointers)
  - hard to make low-fragmentation while keeping list merging cost low

Incremental Tracing Collectors (the organization in the paper is not too good)

- Interleave garbage collection (marking and reclamation) with allocation and program actions (*mutation*)
- Tricolor marking is a useful abstraction
  - reached objects with reached descendants are black
  - reached objects whose descendants are not all reached are grey
  - not-reached objects are white
- **Important invariant**: no black object holds a pointer directly to a white object

Problems and Solutions

- First problem: the mutator may cause violations of the invariant; the GC must notice so as to fix it
- Violation of the invariant would mean that some objects may not be traversed
- For collectors that move objects, a second (easier) problem emerges: the GC is itself a mutator and it can make program data invalid (objects are moved)
- Two kinds of solutions:
  - *read barriers*
  - *write barriers*

Write Barriers

- Trap writes (of pointers to objects)
- Two occasions of interest:
  - writing of a pointer to a white object in black object
  - overwriting of the original pointer to the white object
- Two kinds of write barriers depending on which case they catch:
  - *snapshot-at-beginning*: take a (virtual) snapshot of all pointers to white objects. In reality, trap writes that could be overwriting a pointer to a white object
  - *incremental update*: catch writing in black objects; if pointer to white object is written, revert black object to grey (or make white object grey)

Snapshot-at-beginning

- Prevents overwriting pointers to white objects (and making them seem unreachable)
- When a location is written to, push the overwritten value on a stack, traverse it later
- Fairly conservative: whatever was live at the beginning of collection, will stay live
  - no possibility of making something unreachable *during* the collection
- But only writes to grey and white objects need to be trapped (why?)

Incremental Update

- Notices when a pointer escapes *into* a location that has been traversed (black)
  - all writes to black objects are trapped
- The black object is then made grey (lazy) or the white object pointed to is made grey (eager)
- This ensures that objects can become garbage after the collection has started
- Interesting question: should newly allocated objects be black or white?
  - if black, too conservative (they may die very soon—before the GC is over)
  - if white, complicated. The root set should be re-traversed at the end of a regular GC
- Can the stack be traversed non-atomically?
  - a write barrier that includes it is inefficient

Read Barriers

- The entire "white" space is read protected. Reading white objects makes them turn grey
- Quite expensive without specialized hardware
- Again question is, where to allocate new objects?
- Also, how to make sure that the GC does not run out of space?
  - can tie rate of collection to rate of allocation
- Read barriers can be used with copying collection or implicit non-copying collection. The latter is the *treadmill* technique
  - we now have 4 lists (one extra for allocations that occur during GC)

Generational Garbage Collection

- Most objects die very quickly, but the ones that survive a little tend to survive a lot longer
- Hence, the *generational* idea: multiple generations are GCed independently
  - improves locality
  - reduces work that needs to be done
- Usually 2 generations: the younger one is typically much smaller than the older one
- Main issue: how to treat inter-generational references?

Inter-Generational References

- Liveness is a global property: to tell what is garbage we need to take all generations into account
- Young-to-old references: small problem. Worst case, we traverse young generation (small) every time we GC the old one (rarely)
- Old-to-young references: big problem. Solutions:
  - level of indirection (slows down all pointer accesses from old to young)
  - write barrier on old generation, accounting of all the pointers from it to young
    - can trap all writes. Even better: trap on first write, set dirtyness info, unprotect page (so that subsequent writes are fast), scan dirty pages at end
- Typically many more pointers from new to old