Garbage Collection in an Uncooperative Environment

H. Boehm and M. Weiser

Presented by

Suman Jana

(Some slides courtesy Srilakshmi Pendyala)

Why Think About Uncooperative Environments ?

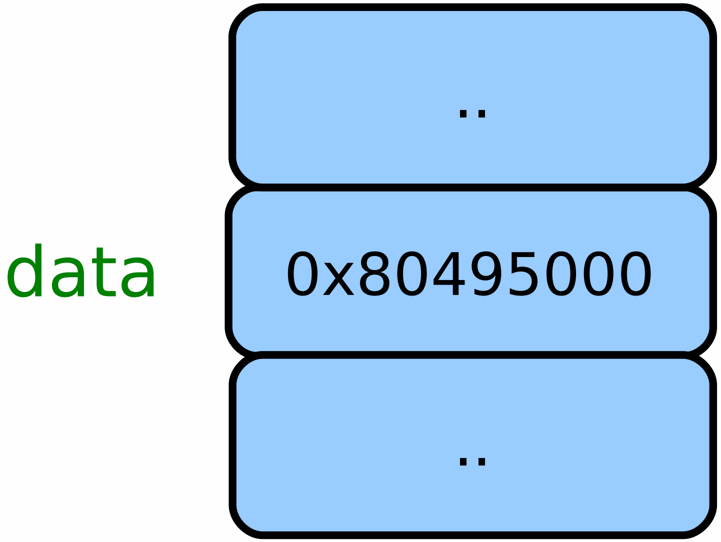
- Programmers do not want to pay for GC
  - GC + manual memory management ?
- GC bookkeeping reduces space for data
  - Tagging integers reduces max integer value
- Partial GC support for existing languages
  - C, Pascal, Russell
- Bugs in read/write barriers are hard to detect

Uncooperative Environment

- Compiler cannot distinguish pointers from data accurately
  - Static analysis ? (dynamic data structures)
  - Has to be conservative
- No read/write barriers
- Possibly no explicit reset of unused references
  - Optimized code can skip clearing stale register content

What can go wrong if data is mistaken as pointer ?

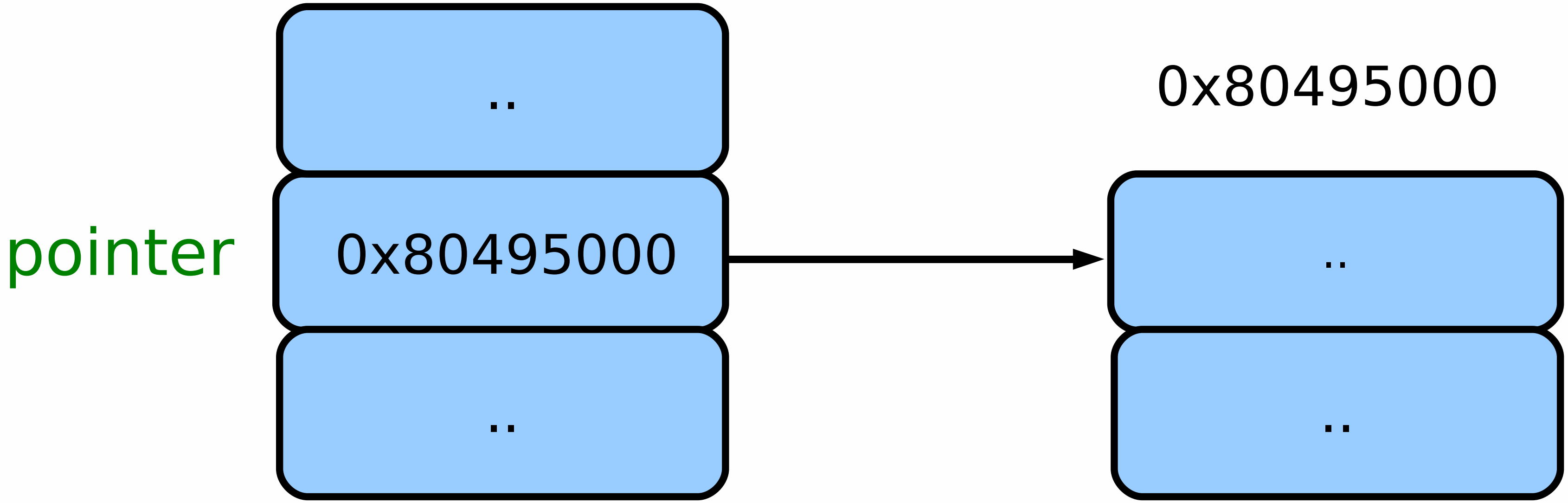
- Incorrect modifications during compact/copy



Mutator view of object 1

What can go wrong if data is mistaken as pointer ?

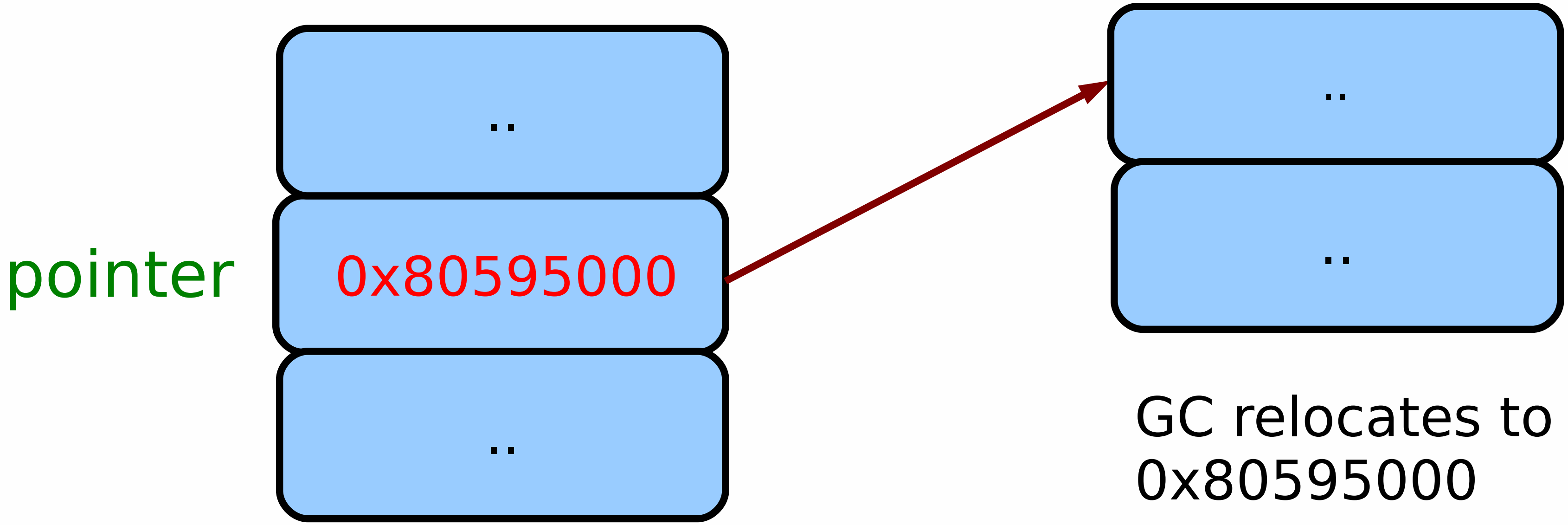
- Incorrect modifications during compact/copy



GC view of object 1

What can go wrong if data is mistaken as pointer ?

- Incorrect modifications during compact/copy

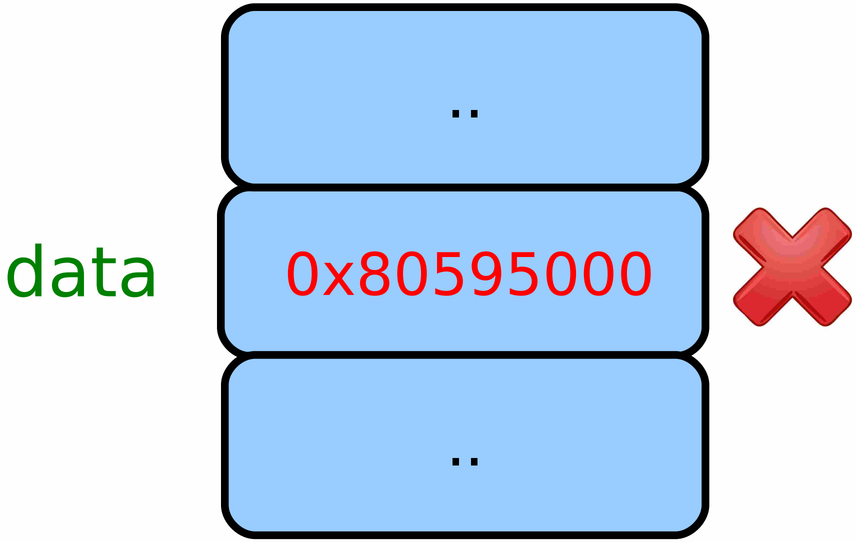


GC relocates to 0x80595000

GC view of object 1

What can go wrong if data is mistaken as pointer ?

- Incorrect modifications during compact/copy



Mutator view of object 1

Conservative GC Assumptions

- Mutator does not intentionally hide references to objects
  - No pointer hiding
- Pointers only point to beginning of objects
  - No interior object pointers (realistic ?)
- No objects greater than 4 KB

Conservative GC details

- Mark-Sweep, Stop-the-World
  - No Copy/compaction
  - Incorrect pointer detection only hurts performance, not correctness
- For marking each data value d in stack & registers call verifyPointer(d)
  - If verifyPointer(d)=TRUE, treat d as pointer
- Less accurate verifyPointer → more memory leak
- Modify memory allocator to improve accuracy of verifyPointer

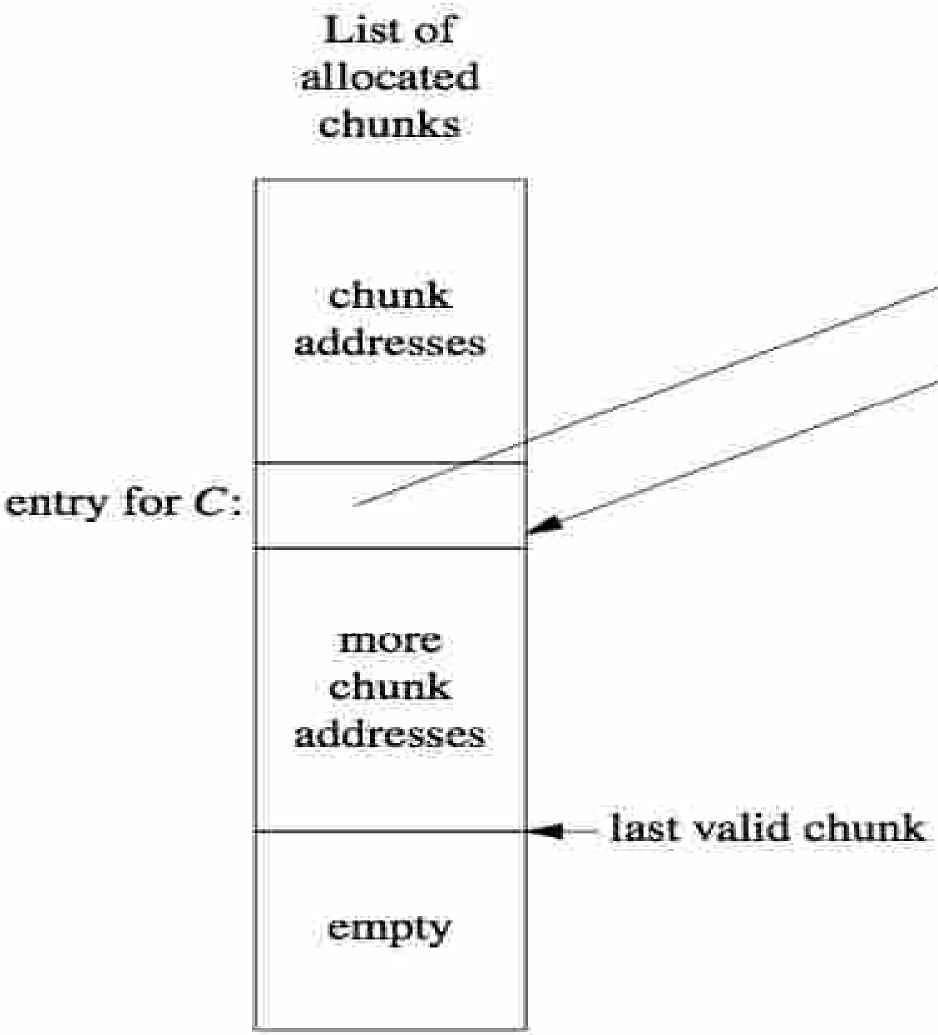
Conservative GC details (cntd.)

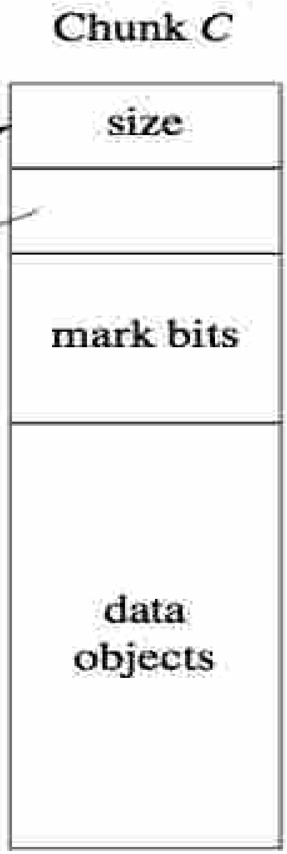
- Modified sweep phase
  - If a object is not marked, add the object to corresponding free-list
  - If an entire chunk is not marked, return the chunk to OS allocator
  - Multiple adjacent free chunks are coalesced and returned

Memory Allocator

- Allocation in 4KB chunks (also 4KB aligned) from OS allocator
- Each chunk contains same-sized objects
- Free-lists for smaller objects
- Dedicated chunk for larger objects
- Global list of allocated chunks
- Each chunk header contains
  - Size of objects in the chunk
  - Pointer to corresponding entry in global list
  - Mark bits for objects

Memory Allocator (cntd.)





verifyPointer(d)

- If (d<lowest heap addr) or (d>highest heap addr) return FALSE
- Find chunk C containing d's target object
  - C = d & 0xffff0000 (why?)
- If C not in Global allocated chunk list return FALSE
- If ((d-C) mod C->object_size ==0) and ((d-C)+object_size <= 4KB) return TRUE
  Else return FALSE

Minimizing verifyPointer False positives

- Process address space follows standard UNIX layout
  - Heap starting address is 0x80***
  - No false positives for small data values
- Separation of "atomic" and "composite" objects
  - "atomic" objects cannot contain any pointers
  - Extra chunk header bit indicates "atomic" objects

Conservative GC issues

- Memory leak
  - Some unused objects may never be collected (data mistaken as pointers to unused objects)
- Difficult to support copy/compaction
  - Modifying data mistaken as pointers will result in incorrect behavior
- Difficult to support concurrent/incremental GC
  - No read/write barriers

Experimental Results

- Russell GC Marking took 1.9 s/MB of accessible memory in heap and sweep phase took 0.4 s/MB on a 25 MHz Sun 3/260
- Successfully ran two large unmodified C programs - TimberWolf and SDI. with GC
  - Re-linked programs to call GC allocator instead of standard Unix allocator.
- Noticed significant fragmentation
  - Free space in a chunk can not be reused for different-sized objects

Experimental results (cntd.)

- Issues with SUNVIEW+GC
  - Dynamically allocated memory remapped to refer to frame buffer: used 'valloc' calls
    - Soln: never free 'valloc' allocated memory
  - Allocated large chunks of memory using "malloc" and divided it into multiple parts for fast allocation, did not keep the original "head" pointer
    - Soln: Recognize such calls and do not free those locations

GC as a Debugging Tool

- GC can identify memory leaks
  - Find not-freed inaccessible allocated memory
- Steps
  - Record function names are recorded in a list for "malloc" call
  - for 'free' call, remove the corresponding list-entry
  - If GC finds any inaccessible object declare it as memory leak along with the corresponding source fucntion from malloc-list

Discussion Questions

- Concurrent conservative garbage collection ?
  - Checkpointing heap at the beginning of mark phase
  - Lazy copying (COW pages) can decrease checkpointing overhead
- Do we really need conservative collectors ?
  - Are the motivations given in the paper justified?