# Conservative GC

# Compiler provides

- Location of references in
  - Roots
  - Objects
- Promise that references are to valid objects
- Write barriers, etc.

# Conservative GC

- Compiler does not tell us locations of (some?) references
- Compiler may not promise references are to beginning of object
- No access barriers whatsoever
- Think: Plain old C

# Plain old C

```
obj = gcAlloc(sizeof(…));
for (i = 0; i < ULONG_MAX; i += 16000000) {
  obj->x = i;
  …
}
```

- What we see is unknown data.
- Non-references may look like references.

# Plain old C

- Compiler can optimize arbitrarily
- e.g., if `obj->x` access is repeated, store location of `obj->x` instead of `obj` (interior pointers)
- No yield-points, so references can be anywhere

# What we *do* have

- Binary interface per platform defines:
  - Where stack is, how it grows
  - Which registers have arbitrary data
  - Alignment
- This gives us just enough to make educated guesses

# What we control

- We still have control of the GC’s domain:
  - Allocation, and thus object headers
  - When to perform collection
  - Stop-the-world is possible
- Can still store enough to parse heap, etc.

# How to use it

- “If it looks like a reference, let’s assume it’s a reference.”
- We can walk through the stack like an array (alignment is guaranteed)
- False references are tolerable, false non-references are not

```
handle(maybeReference):
  if inGCPool(maybeReference):
    obj := findObjectBase(maybeReference)
    if obj == NULL: return
    if !obj->header.mark:
      scan(obj)
      obj->header.mark := 1
```

# In GC pool

- Given a value, could it be a reference to a pool?
- If `(x >= pool->start && x < pool->end)`, yes
- Option one: Check by checking every pool

```
inGCPool(maybeReference):
  foreach pool p:
    if (maybeReference >= p->start &&
        maybeReference < p->end):
      return true
  return false
```

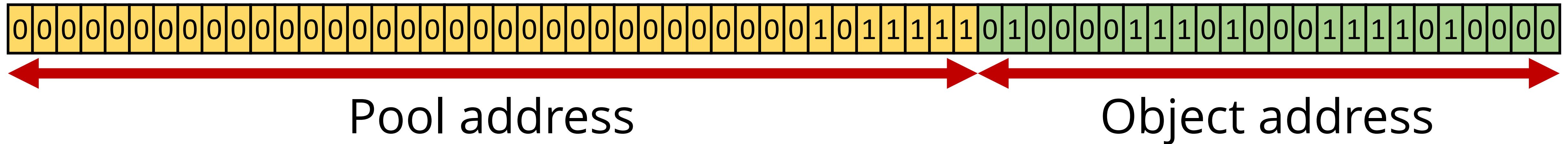
This algorithm is 100% precise,
and 100% slow.

# In GC pool

- Ultimately we must be 100% correct, but
- need a faster way to separate the definitely-not-references from the maybe-references
- Usual technique: Bloom filter

# Bloom filter

- Remember:




- Hash pool bits to get index into bit-array
- Bit-array set to 1 for valid pools

# Bloom filter example

- Example (stupid) hash function:

```
unsigned char hash(addr) {
    return (addr>>POOL_BITS)&0xFF;
}
```

- Need a bit-array of size 256 bits

# Bloom filter example

Example object:

Hash

0000000000000000000000000000000001011111 | 010000111010001111010000

Pool address | Object address

There is a pool with hash
0b01011111 == 0x5F

Bloom filter table:

0x5F

000000000000000000000000000000000000000100000000000000000000000000000

```
inGCPool(maybeReference):
  if bloomFilter[hash(maybeReference)]:
    foreach pool p:
      if (maybeReference >= p->start &&
          maybeReference < p->end):
        return true
  return false
```

This algorithm is still precise, much faster,
but still involves a linear search over pools.

# Pool lookup

- Just one more trick: Store pools as a tree for O(log n) lookup
- Bloom filter rejects most potential pointers, O(log n) pool lookup gets perfect precision

# Hash function

- Bloom filter requires a hash function
- Hash function should be fast
- Range of hash function determines bitmap size
- Between 10-bit and 16-bit reasonable (1024-bit bit-array to 65536-bit bit-array)

# Hash function

- Alignment matters: We can ignore bits within a pool
- Lower bits tend to be more valuable than higher
- Typical choice might be:
  2-bit hash of top 32-bits,
  concatenate next 8-bits

# Object lookup

- We may have interior pointers
- We may still have *false* pointers!
- Card parsing can handle the first, partially handle the second

```
findObjectBase(maybeReference):
  pool := poolOf(maybeReference)
  if maybeReference > pool->free:
    return NULL
  cardNo := cardOf(maybeReference)
  obj := firstObjInCard(pool, cardNo)
  while !obj or obj > maybeReference:
    cardNo := cardNo - 1
    obj := firstObjInCard(pool, cardNo)
  while maybeReference > obj + obj->header.size:
    obj := obj + obj->header.size
  if obj is a free object:
    return NULL
  return obj
```

# Observation

- Things get even more complicated with large objects (pool alignment imperfections)
- Because false references are possible, moving is impossible
- Remember: Until you’re confident it’s a reference, you can’t read what it points to

```
collect():
  worklist := new Queue
  for loc in stack:
    if inGCPool(*loc):
      worklist.push(loc)
  for loc := worklist.pop()
    obj := findObjectBase(*loc)
    if obj == NULL: continue
    if !obj->header.mark:
      for loc in obj to obj + obj->header.size:
        if inGCPool(*loc):
          worklist.push(loc)
      obj->header.mark := 1
  (sweep as usual)
```

# Breather

- By conservatively guessing anything that looks like a reference is a reference, can GC C
- e.g. libgc does this
- Many costs, plus false positives

# Thoughts

- If program sets a field of an object, compiler cannot usually optimize that away or generate interior pointers
- We have more flexibility for object fields than for roots
- Gives rise to *partially conservative GC*

# Idea

- Demand a precise heap, conservative stack
- Compiler now must tell us the shape of heap objects, but has free reign over stack
- Roots are usually smaller than heap, so saves expense for that part

# Classes

- This gives us two classes of references:
  - Conservative (root) reference
  - Precise (heap) reference
- That gives us two classes of object during GC:
  - Object with conservative references (cannot move)
  - Object with only precise references (could move)

# Moving

- Partially-conservative GC makes moving possible again
- Need a notion of "pinned" objects: Cannot move because of root references
- Any unpinned objects may still move

# Bartlett’s algorithm

- Scan roots to *pin* objects
- Scan pinned objects to *copy* unpinned objects
- Sweep to create free objects between pinned objects
  - Pinned objects are rare, little fragmentation

```
bartlettSemispaceCollect():
  pinlist := new Queue
  worklist := new Queue
  fromspace, tospace :=
    tospace, fromspace
  pinsize := 0
  foreach loc in stack:
    if inGCPool(loc):
      pinlist.push(loc)
  while loc := pinlist.pop():
    obj := findObjectBase(*loc)
    if obj == NULL: continue
    if !obj->header.mark:
      (scan obj into worklist)
      obj->header.mark := 1
      pinsize := pinsize +
        obj->header.size
  tospace->minimumFree := pinsize
  while loc := worklist.pop():
    obj := *loc
    if !obj->header.mark and
       !obj->header.forward:
      if obj !in tospace:
        (copy obj to newObj in
         tospace)
        (scan new obj)
        obj->header.forward :=
          newObj
      else:
        (scan obj)
        obj->header.mark := 1
  (sweep to create free-list for
   tospace)
```

# Bartlett’s + generational

- Only possible with write barrier
- Unpinned objects can be promoted as usual
- Pinned objects *can* be promoted
- A generation a list of pools
- Simply move a pool with pinned objects from young to old generation

# Pool moving

- Certain objects will be pinned forever: Keeping them young is a bad idea
- Moving pools is a last resort
- If done too often, old generation gets quite large
- Should have a way to move back, if objects become unpinned and die

# False positives

- False references are uncommon in real code
- They are a security concern: Malicious code can intentionally pin objects if it can guess addresses
- Practical implementations prove this attack is rare

# Conservative GC thoughts

- Conservative is a huge price
- Because roots are small, partially conservative is a much smaller price
- Partially conservative collection very attractive for language implementations (e.g. WebKit's JavaScript)
- Full conservative usually just a stopgap

[but still better than reference counting]