Mark-Sweep and Mark-Compact GC

Richard Jones
Anthony Hoskins
Eliot Moss

Presented by Pavel Brodsky
04/11/14

Our topics today

- Two basic garbage collection paradigms:
  - Mark-Sweep GC
  - Mark-Compact GC

Definitions

- **Heap** - a contiguous* array of memory words.
- **Granule** - the smallest unit of allocation (usually a word or a double word) in the heap.
- **Roots** - objects in the heap, directly accessible by the application code.

Definitions (cont.)

- A **collector** is the thread(s) responsible for garbage collection.
- **Mutators** are threads that alter the heap (application code).

Notation

- $\leftarrow$ is the assignment operator.
- = is the equality operator.
- `Pointers(obj)` - all of `obj`'s fields (which might be data, objects, or pointers to objects).

Assumptions

- In our code, possibly multiple *mutator* threads, only one *collector* thread.
- **Stop-the-world** assumption - all *mutator* threads are stopped when the *collector* thread runs:
  - Simulates **atomicity**.

Liveness

- An object is **live** if it will be accessed by a *mutator* at some point in the future.
- It's **dead** otherwise.
- True **liveness** is undecidable, so we turn to an approximation - **pointer reachability**.

Pointer reachability

- In our context:
  - An object is *live* iff it can be reached by following a chain of references from the *roots*.
  - An object is *dead* iff it cannot be reached by any such chain.
- A safe estimate - all dead objects are certainly dead (cannot be brought back to life by *mutator* threads).

The tricolor abstraction

- A convenient way to describe object states:
  - Initially, every node is white.
  - When a node is first encountered (during tracing), it is colored **grey**.
  - When it has been scanned, and its children identified, it is colored **black**.
- At the end of each sweep, no references from **black** to white objects.
  - All the white objects are unreachable = *garbage*.

Mark-Sweep (McCarthy, 1960)

- Two main phases:
  - **Tracing/Marking**:
    - Traverse the graph of objects from the *roots*
    - Follow pointers
    - *Mark* each encountered object.
  - **Sweeping**:
    - Examine every object in the heap
    - Reclaim the space of any unmarked object (*garbage*).

Bitmap marking

- Use a **mark-bit** to determine the *liveness* of an object (1 = *live*, 0 = *dead*).
- Two ways to keep track of *mark-bits*:
  - A bit in the header of an object.
  - A **bitmap**.

Mark-Sweep (cont.)

- An **indirect** collection algorithm:
  - Doesn't identify garbage.
  - Identifies all the live objects.
    - Concludes that all the rest is garbage.
- Recalculates the **live set** (a set of all the *marked*/*live* objects) with each invocation.
- *Whiteboard example*.

New

GC's interaction with the *mutator*:

```
New():
    ref <- allocate()                   // Mutator thread tries
                                        // to allocate memory

    if ref = null                       // If the heap is full
        collect()                       // Use GC
        ref <- allocate()               // Try to allocate again

        if ref = null                   // If the heap is still full
            error "Out of memory"       // Return an error

    return ref

atomic collect():
    markFromRoots()                     // Mark all live objects
    sweep(HeapStart, HeapEnd)           // Remove all unmarked
```

markFromRoots

```
markFromRoots():
    initialise(worklist)                    // Start with an empty worklist

    for each fld in Roots                   // For each root object
        ref <- *fld
        if ref != null && !isMarked(ref)    // If it isn't marked
            setMarked(ref)                  // Mark it
            add(worklist, ref)              // Add it to the worklist
            mark()                          // Mark all its children

initialise(worklist):
    worklist <- empty
```

Note: `mark()` doesn't have to be called after adding every root object. Its call can be moved outside the loop.

mark

```
mark():
    while !isEmpty(worklist)                    // As long as the worklist isn't empty
        ref <- remove(worklist)                 // Remove some object from the list
        for each fld in Pointers(ref)           // Check its every field (child)
            child <- *fld
            if child != null && !isMarked(child)    // If it isn't marked
                setMarked(child)                // Mark it
                add(worklist, child)            // And add it to the worklist
```

- Worklist implementation:
    - A single-thread *collector* can be implemented with a stack.
    - Meaning - the traversal is done using DFS.

Correctness of `mark`

- Termination is enforced by not adding already marked objects to the *worklist*.
- Eventually, the list becomes empty.
- At that point, every object reachable from the *roots* has been visited and was marked.

sweep

```
sweep(start, end):
    scan <- start                       // Start from the beginning
    while scan < end                    // Progress until the end of the list
        if isMarked(scan)               // Unmark every marked object
            unsetMarked(scan)
        else
            free(scan)                  // Free all those that weren't marked

        scan <- nextObject(scan)        // Continue to the next object
```

- Reminder: we call `sweep` from `collect` with HeapStart and HeapEnd as the parameters.
- We then traverse the whole heap, and reclaim the space of any unmarked object.

Possible issues with mark-sweep

- Severe **fragmentation** (caused by not moving objects).
- Heap traversal in the presence of **padding**.

Improving Mark-Sweep

- **Linear bitmaps** [Printezis and Detlefs, 2000]
- **Lazy sweeping** [Hughes, 1982]
- If there's time:
  - **FIFO prefetch buffer** [Cher *et al*, 2004]
  - **Edge marking** [Garner *et al*, 2007]

Bitmaps

- A *bitmap* is a table of *mark-bits*.
- Each bit corresponds to an object on the heap.
- Fast access (may be held in the RAM).
- Can find the corresponding bit in O(1) time.

MS Improvement: Linear bitmaps

- Use *bitmaps* to reduce the amount of space used for the *mark stacks* (the *worklist*):
  - Mark all the root objects in the *bitmap*.
  - Next, **linearly** traverse the *bitmap*, top down, and only add new children to the *worklist* if they are *below* a “finger”.
- Maintain the invariant that marked object below the “finger” are **black**, and those above it are **grey**.

Bitmap mark

```
mark():
    cur <- nextInBitmap()                          // Get the next object
    while cur < HeapEnd                            // Linearly move through the heap
        add(worklist, cur)                         // Process the current object
        markStep(cur)                              // Invoke markStep
        cur < nextInBitmap()

markStep(start):
    while !isEmpty(worklist)                       // Process the whole worklist
        reg <- remove(worklist)                    // Remove some object from the list
        for each fld in Pointers(ref)              // Check its every field (child)
            child <- *fld
            if child != null && !isMarked(child)   // If it isn't marked
                setMarked(child)                   // Mark it
                if child < start                   // If it's above the "finger"
                    add(worklist, child)           // Add it to the worklist
```

- Main change is in the highlighted row: new objects are only added to the worklist if they are above the current “finger”.
- Possibly a constant improvement in running time (not asymptotic).

MS Improvement: Lazy sweeping

- **Motivation**: reduce (or even eliminate) *mutators* stop time during the sweep phase.
- Two **observations**:
  a. Once an object is *garbage*, it remains *garbage*: it can neither be seen nor resurrected by a *mutator*.
  b. *Mutators* cannot access *mark-bits*.
- **Conclusion**: the sweeper can be executed in **parallel** with *mutator* threads.

Lazy sweeping

- Amortise the cost of sweeping by having the *allocator* perform the sweep.
- `allocate` advances the sweep pointer until it finds sufficient space.
  - Usually more practical to sweep a block at a time.

collect and allocate

```
atomic collect():
    markFromRoots()                        // Mark the live objects
    for each block in Blocks               // Decide for each block:
        if not isMarked(block)             // If there are no marked objects in this block
            add(blockAllocator, block)     // return the block to the block allocator
        else
            add(reclaimList, block)        // O/w, queue it for lazy sweeping

atomic allocate(sz):
    result <- remove(sz)                   // Try to allocate size(sz) memory
    if result = null                       // If no free slots of this size are available
        lazySweep(sz)                      // Sweep a bit
        result <- remove(sz)               // And try to allocate again
    return result
```

- Note: blocks are grouped by their size class (*sz*).

lazySweep

```
lazySweep(sz):
    repeat                                           // Do for every block in reclaimList
        block <- nextBlock(reclaimList, sz)          // Get a block of sz's size-class
        if block != null
            sweep(start(block), end(block))          // Sweep the block
            if spaceFound(block)                     // If we were able to sweep enough
                return                               // We're done
    until block = null
    allocSlow(sz)                                    // If we couldn't find a suitable block
                                                     // Move to the slow allocator

allocSlow(sz):
    block <- allocateBlock()                         // Get a block from the block allocator
    if block != null
        initialise(block, sz)
```

Lazy sweep benefits

- Good locality:
  - Object slots tend to be used soon after they are swept.
- Complexity is now proportional to the size of the **live data** in the heap (as opposed to the **whole** heap).
- Performs best when most of the heap is empty.

Bonus: Snapshot mark-sweep

- The basic mark and sweep algorithm stops all *mutator* threads during both *mark and sweep* phases.
- Use the observation that the set of unreachable objects does not shrink
  - So the only time *mutator* threads *must* be stopped is during the mark phase.

Snapshot mark-sweep (cont.)

- Basic algorithm:
  - Stop all *mutators*
  - Take a *snapshot* (replica) of the heap and *roots*
  - Resume *mutators*
  - Trace the replica
  - Sweep all objects in the original heap whose replicated counterparts are unmarked.
    - These objects must have been unreachable at the time the snapshot was taken.
    - They will remain unreachable until the collector frees them.

Snapshot mark-sweep (limitations)

- The problem with this approach is that making a snapshot of the heap is not realistic.
  - Requires too much space and time.
- Usually, only a small part of the heap it is modified at a time.
- A full solution to this problem exists, but is outside the scope of this discussion.

Mark-sweep GC advantages

- **Low overhead**:
  - Basic **mark-sweep** imposes no overhead on *mutator* `read` and `write` operations.
- **Good throughput**:
  - Setting a bit or byte is cheap
  - The mark phase is very inexpensive.
- **Good space usage**:
  - A single bit/byte for an object is an inexpensive way to store that object's state.

Moving objects

- The benefit in not moving objects is that **mark-sweep** is suitable for use in environments with no type-safety.
  - Moving an object forces us to update the roots.
- The disadvantage is severe *fragmentation*, especially for long-running programs.

Possible solution to not moving

- Some collectors that use **mark-sweep**, also periodically employ another algorithm, such as **mark-compact**, to defragment the heap.
- Especially useful in cases where the program doesn’t use consistent object sizes.

Mark-compact GC

- **Two main phases:**
    - **Tracing/marking**:
        - Mark all the *live* objects.
    - **Compacting**:
        - Relocate *live* objects
        - Update the pointer values of all the *live* references to objects that were moved.
    - The number/order of passes and the way in which objects are relocated varies.

Compaction order

- Three ways to rearrange objects in the heap:
  - **Arbitrary**: objects are relocated without regard for their original order.
    - Fast, but leads to poor spatial locality.
  - **Linearising**: objects are relocated to be adjacent to related objects (siblings, pointer and reference, etc.)
  - **Sliding**: objects are slid to one end of the heap, “squeezing out” garbage, and maintaining the original allocation order in the heap.
    - Used by most modern **mark-compact** collectors.

Mark-compact GC

- The **compacting** technique minimizes (or even eliminates) *fragmentation*.
- Very fast, sequential allocation:
  - Test against a heap limit.
  - ‘Bump’ a free pointer by the size of the allocation request.
- We only discuss *in-place* compaction (as opposed to copying collection).

The algorithms we will discuss

- **Edward's Two-finger compactions** [Saunders, 1974]
- **Lisp 2 collector**
- **If there's time:**
  - **Threaded compaction** [Jonkers, 1979]
  - **One pass algorithms** [Abuaiadh *et al*, 2004, Kermany and Petrank, 2006]

Invocation

- All compaction algorithms are invoked as follows:

```
atomic collect():
    markFromRoots()        // Mark all live objects, starting from the roots
    compact()              // Compact the heap
```

Two-finger compaction

- A two-pass, **arbitrary** order algorithm
- Works best if objects are of a fixed size.
- Basic idea:
  - Given the number of live objects,
  - Set a **high-water mark**:
    - Move all the live objects into gaps below it.
    - Reclaim all the space above it.

TF compaction: `relocate`

```
compact():
    relocate(HeapStart, HeapEnd)                    // Compact the objects
    updateReferences(HeapStart, free)               // Update the pointers

relocate(start, end):
    free <- start                                   // Start at the beginning
    scan <- end                                     // Start at the end

    while free < scan
        while isMarked(free)                        // Advance until a gap is found
            unsetMarked(free)                       // Unmarking passed live objects
            free <- free + size(free)

        while !isMarked(scan) && scan > free        // Retreat until a live object is found
            scan <- scan - size(scan)

        if scan > free                              // Move the live object into the gap
            unsetMarked(scan)
            move(scan, free)                        // Move the data
            *scan <- free                           // Leave a forwarding address
            free <- free + size(free)               // Advance free
            scan <- scan - size(scan)               // Retreat scan
```

The **forwarding address** will allow us to update old values of pointers to objects above the *high-water mark* (that `free` is pointing to, at the end of the first pass).

TF compaction: `updateReferences`

```
updateReferences(start, end):
    for each fld in Roots                       // Update the roots
        ref <- *fld                             // If the child is pointing beyond the
        if ref >= end                           // high-water mark
            *fld <- *ref                        // Point it to the forwarding address

    scan <- start
    while scan < end                            // Update all the fields in the live
        for each fld in Pointers(scan)          // region the same way
            ref <- *fld
            if ref >= end
                *fld <- *ref
        scan <- scan + size(scan)
```

TF compaction: pros and cons

- Pros:
  - **Simplicity and speed**: minimal work is done at each iteration.
  - **No memory overhead**: forwarding addresses are written into slots above the *high-water mark*, *after* the information has been moved, so no information is ever destroyed.
- Cons:
  - The movement of `scan` requires the ability to traverse the heap *backwards*.
  - Arbitrary move order.

TF compaction: an improvement

- A possible improvement to the *mutator* locality is to move **groups** of consecutive live objects into large gaps, using the fact that objects tend to live and die together in clumps.

Lisp 2

- A **sliding** collector algorithm.
- Adds a field to the header of every object for the **forwarding address**.
  - Can also be used for the *mark-bit*.
  - That memory overhead is the **chief drawback** of the algorithm.
- Can be used with objects of varying sizes.
- Arguably the fastest compaction algorithm.

Three passes over the heap

- The first pass (after marking):
  - Compute the future location of each live object.
  - Store it in the object's `forwardingAddress` field.
- The second pass:
  - Updates all pointers to the new *forwarding address*.
- The third pass:
  - Moves the actual objects to the *forwarding address*.

Pass direction

- The direction of the passes (upward in the heap) is opposite to the object’s moving direction (downward).
- This guarantees that when the object is copied (in the third pass), the location is already vacant.

computeLocations

```
compact():
    computeLocations(HeapStart, HeapEnd, HeapStart) // Compute the forwarding addresses
    updateReferences(HeapStart, HeapEnd)            // Update the roots and references
    relocate(HeapStart, HeapEnd)                    // Move the objects

computeLocations(start, end, toRegion):
    scan <- start                                   // A finger to iterate over all objects
    free <- toRegion                                // A finger to indicate the next address to use
    while scan < end
        if isMarked(scan)                           // If it's alive, update its forwarding
            forwardingAddress(scan) <- free         // address to the next one free points to
            free <- free + size(scan)               // Advance free
        scan <- scan + size(scan)                   // Move to the next object
```

Ignore any dead objects - no need to relocate them.

updateReferences

```
updateReferences(start, end):
    for each fld in Roots                          // Update the roots
        ref <- *fld
        if ref != null
            *fld <- forwardingAddress(ref)         // Point to the object's forwarding address

    scan <- start                                  // Do the same for all the live objects
    while scan < end
        if isMarked(scan)                          // If the objects is live
            for each fld in Pointers(scan)         // Update its every field (child) to point
                if *fld != null                    // at the correct address (as stored in the
                    *fld <- forwardingAddress(*fld) // header of the object it's pointing to)
        scan <- scan + size(scan)
```

Use the *forwarding addresses* to update the references of the *live* objects.

relocate

```
relocate(start, end):
    scan <- start                                    // Start at the beginning
    while scan < end                                 // Traverse the whole heap
        if isMarked(scan)                            // For every live object
            dest <- forwardingAddress(scan)          // Get its destination address
            move(scan, dest)                         // Move the object itself there
            unsetMarked(dest)                        // And unmark it, to prepare for
        scan <- scan + size(scan)                    // the next invocation of compact
```

Move every object to the *forwarding address*.

Mark-compact collection: pros

- Compaction is very effective way to deal with heap *fragmentation*.
- Allows for very fast sequential allocation, after compaction.
- Effective in the case of long lived (or immortal) objects, that remain unmoved at the bottom of the heap.

Mark-compact collection: cons

- Has some space overheads incurred by storing *forwarding addresses*.
- Usually has a slower throughput than **mark-sweep** or **copying** GC, as it requires more passes over the heap.

Discussion

- In MS, think of a way to prefetch objects ahead of time.
- In MS, think of a way to reduce the size of the worklist.
- In MC, think of a way to not use any extra space.
- In MC, think of a way to sweep in one pass.

Fin

＼(^o^)／

Questions?

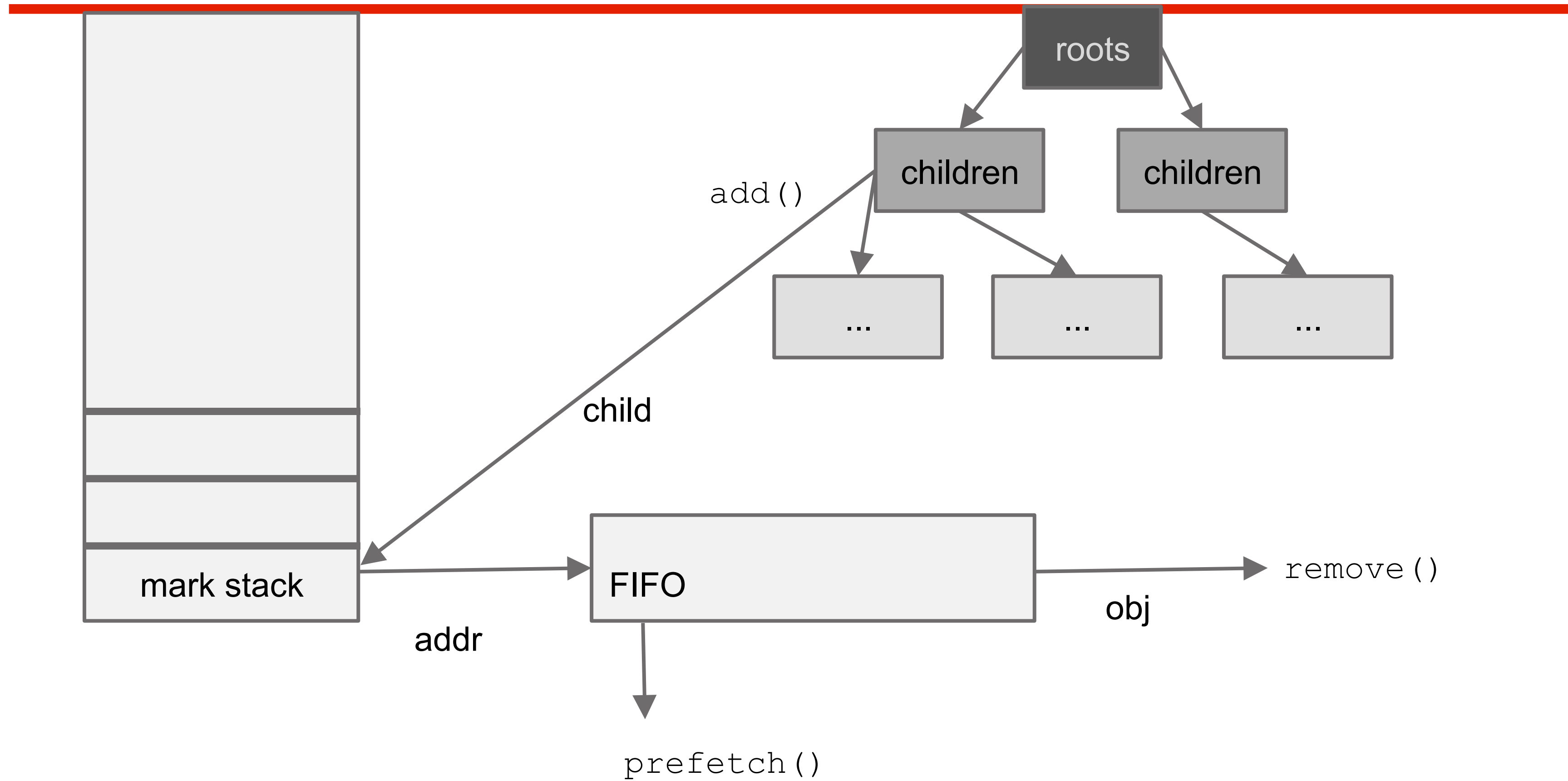
Appendix

Additional algorithms

MS Improvement: FIFO prefetch buffer

- Use a FIFO buffer alongside the mark stack:
  - To add an object to the worklist:
    - Push it onto the mark stack.
  - To remove an object from the worklist
    - Remove the oldest item from the buffer.
    - Insert the entry at the top of the stack to the buffer.
    - Prefetch the object to which the entry points.
      - It will be in the cache when the entry leaves the buffer.

FIFO prefetch buffer



Marking with a FIFO prefetch buffer

```
add(worklist, item):
    markStack <- getStack(worklist)          // Get the current mark stack
    push(markStack, item)                    // Push the new item to its top

remove(worklist):
    markStack <- getStack(worklist)          // Get the current mark stack
    addr <- pop(markStack)                   // Remove the topmost item
    prefetch(addr)                           // Add the object it points to to the cache
    fifo <- getFifo(worklist)                // Get the current FIFO buffer
    prepend(fifo, addr)                      // Add addr to the end of the line
    return remove(fifo)                      // Return the object at the front of the line
```

MS Improvement: Edge marking

- **Motivation**: reduce the number of cache misses when checking `isMarked(child)` during `mark`.
- Add to the worklist every child of an **unmarked** object, without checking.
- Works *in conjunction* with the FIFO buffer.

`mark` with *edge* marking

```
mark():
    while not isEmpty(worklist)                         // Traverse the whole worklist
        obj <- remove(worklist)                         // Remove some object
        if not isMarked(obj)                            // Make sure it isn't marked
            setMarked(obj)                              // Mark it
            for each fld in Pointers(obj)               // Add every one of its non null
                child <- *fld                           // children to the worklist
                    if child  != null
                        add(worklist, child)
```

- We aren't checking whether `isMarked(child)`, but instead adding every child, regardless.
- `isMarked` and `Pointers` now operate *only* on `obj`, which has been (hopefully) prefetched using the FIFO queue, thus, much fewer misses should occur.

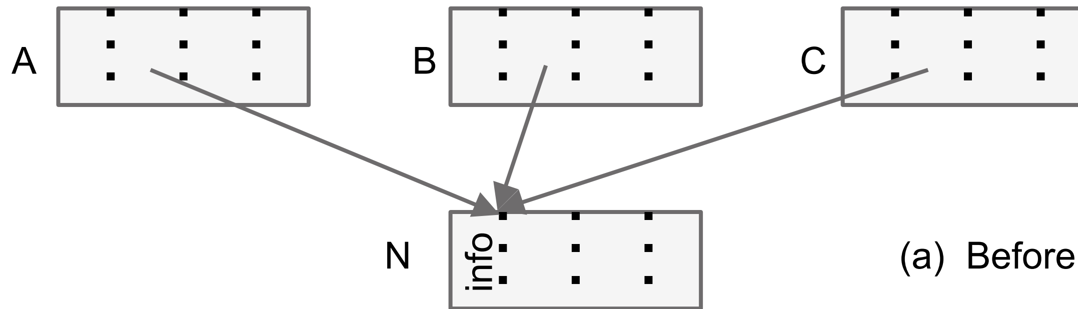
MC improvement: Threaded compaction

- We allow all references to a node N to be found from N.
- Achieved by temporarily *reversing* the direction of pointers.
- The algorithm we discuss is by Jonkers [1979]
- Two passes over the heap:
  - The first to thread references that point forward in the heap.
  - The second to thread backward pointers.

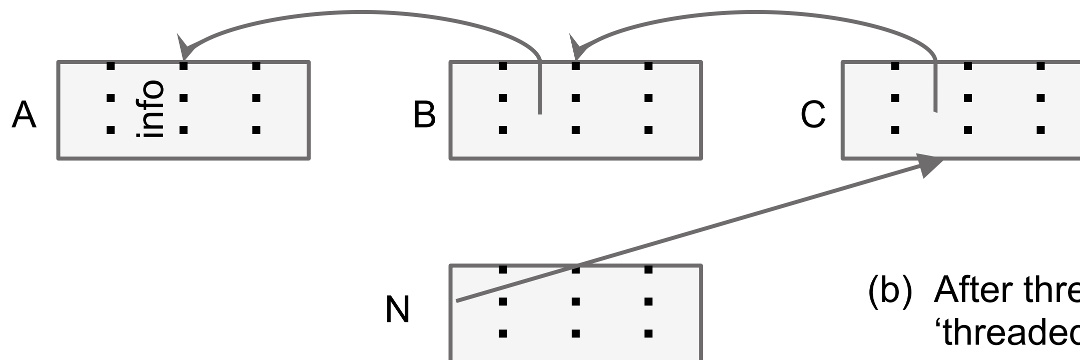
Threaded compaction (cont.)

- **Threading** requires no extra storage yet supports sliding compaction.
- Requires enough room in the header to store an address (a weak requirement).
- Also requires the ability to differentiate pointers from other values (may be harder).

Threading: visualisation



(a) Before threading: three objects refer to N.



(b) After threading: all pointers to N have been 'threaded' so that the objects that previously referred to N can now be found from N. The value previously stored in the header word of N, which is now used to store the threading pointer, has been (temporarily) moved to the first field (in A) that referred to N.

compact, thread and update

```
compact():
    updateForwardReferences()
    updateBackwardReferences()

thread(ref):
    if *ref != null
        *ref, **ref <- **ref, ref
```

```
update(ref, addr):
    tmp <- *ref
    while isReference(tmp)
        *tmp, tmp <- addr, *tmp
    *ref <- tmp
```

- *Illustration on the board.*

updateForwardReferences

```
updateForwardReferences():
    for each fld in Roots                       // Thread the objects Roots point to
        thread(*fld)

    free <- HeapStart
    scan <- HeapStart

    while scan <= HeapEnd                       // Traverse all the objects in the heap
        if isMarked(scan)                       // If it's marked
            update(scan, free)                  // Unthread it
            for each fld in Pointers(scan)      // Thread all its children
                thread(fld)

            free <- free + size(scan)           // Advance free
        scan <- scan + size(scan)               // Advance scan
```

- **Unthreading** means pointing all the threaded objects that used to point to `scan` to the address occupied by `free`, where the object that resides in `scan` will eventually move to.

updateBackwardReferences

```
updateBackwardReferences():
    free <- HeapStart
    scan <- HeapStart

    while scan <= HeapEnd                           // Traverse all the objects in the heap
        if isMarked(scan)                           // If it's still marked after the first pass
            update(scan, free)                      // Unthread it
            move(scan, free)                        // And move the object

            free <- free + size(scan)               // Advance free
        scan <- scan + size(scan)                   // Advance scan
```

- *Threading* of all the children and forward references unthreading were done in the first pass.
- All the *backward* references were *threaded* during the first pass, and these are the only ones we're updating now.

Threaded compaction: pros and cons

- Pros:
  - Doesn't require any additional space
- Cons:
  - Each pointer is modified twice (*thread*/*unthread*).
  - Cache unfriendly - requires chasing pointers (3 times in Jonkers' algorithm: *mark*/*thread*/*unthread*).
  - Object fields must be large enough to hold a pointer.
  - Pointers must be distinguishable from a normal value.

MC improvement: One-pass algorithms

- **Motivation**: perform sliding compaction in one pass.
- Achieved by using a *bitmap*, and another table, to store *forwarding addresses*.

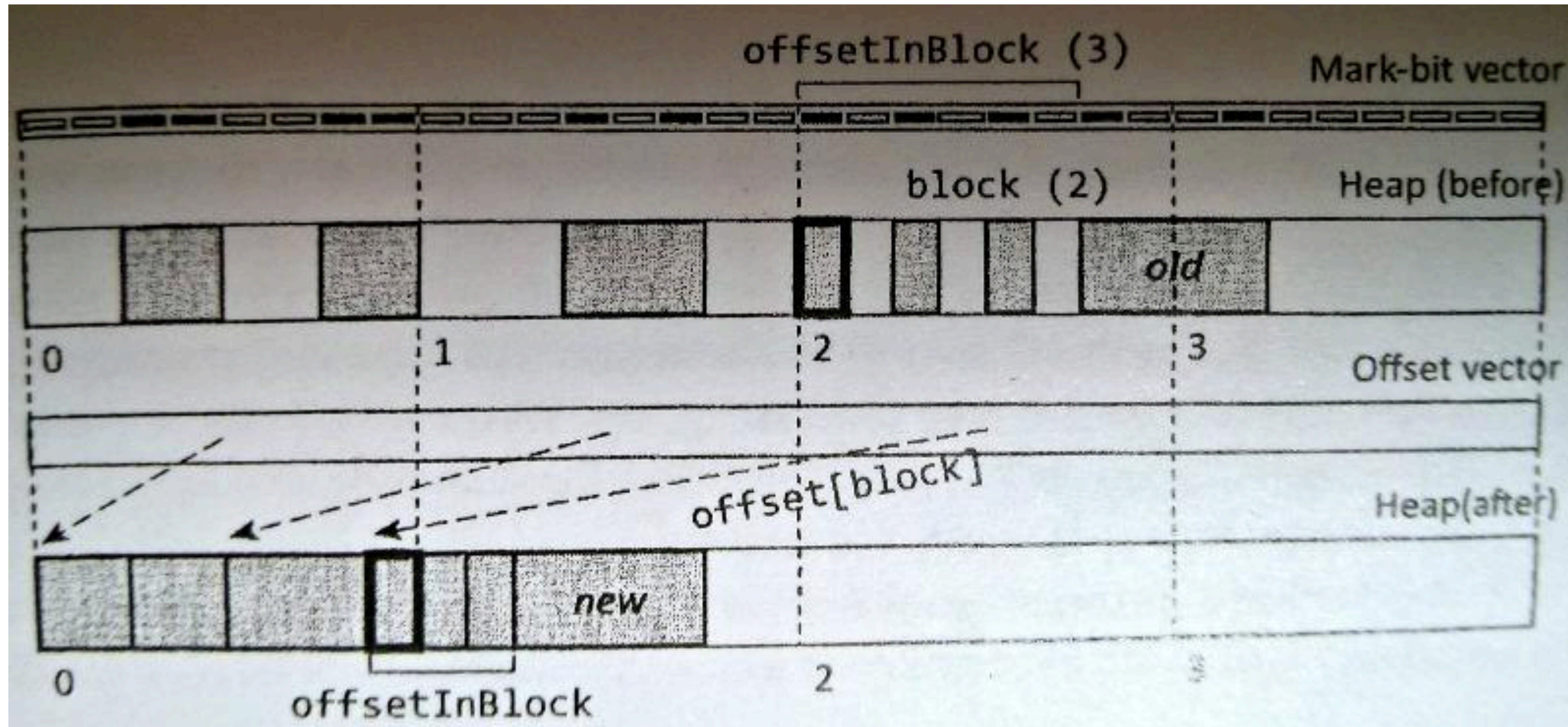
The two tables

- A *bitmap*:
  - One bit for each granule of the heap.
  - Marking sets bits corresponding to the first *and* last granules of each live object.
- An **offset** table for the *forwarding addresses*:
  - Divide the heap into small, equal-sized blocks (256 or 512 bytes).
  - Store the forwarding address of the **first** *live* object in each block in the table.

Address derivations

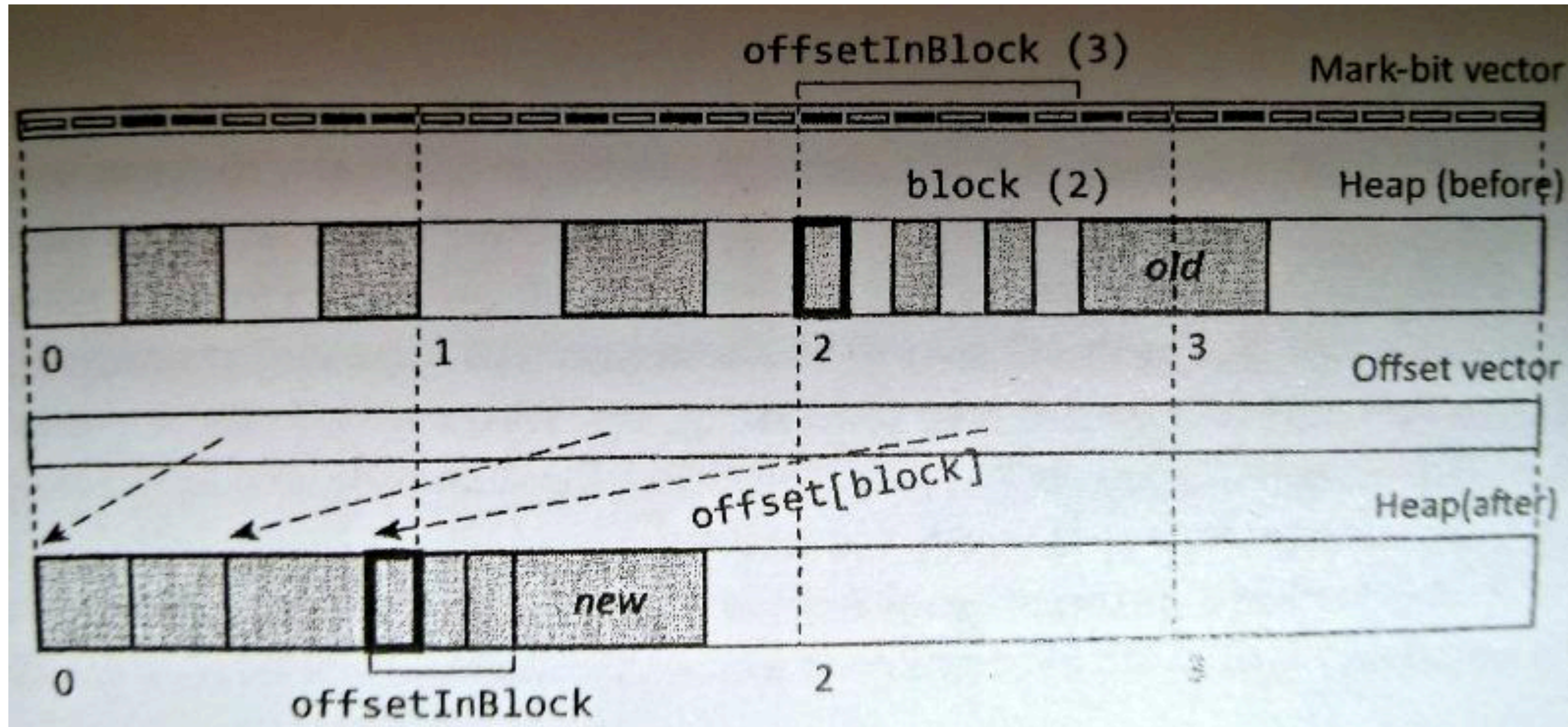
- The new location of the other live objects in a block (other than the first one) can be computed on-the-fly from the offset and mark-bit vectors.
- Similarly, given a reference to any object, we can compute its block number, and thus derive its forwarding address from the entry in the `offset` table and the mark-bits.

Visualization



Consider the object in bold. Bits 2-3, 6-7 in the first block and 4-6 in the second are set. Thus, 7 granules are already taken by objects that come before. So the first live object in block 2 will be relocated to the 8th slot in the heap (as seen in the `offset` vector - see the arrow).

Visualization (cont.)



Consider the object *old*. We obtain its block number and use it as an index into the `offset` vector. This is the address of the first object in the block. Then look at the bitmap, to find the offset in this block (3), and calculate the final address: `offset[block]=7` plus `offsetInBlock(old)=3` equals 10.

computeLocations

```
compact():
    computeLocations(HeapStart, HeapEnd, HeapStart)
    updateReferencesRelocate(HeapStart, HeapEnd)

computeLocations(start, end, toRegion):       // Produce the offset vector from the bitmap
    loc <- toRegion                           // Start at the beginning of the heap
    block <- getBlockNum(start)               // Start from the first block
    for b <- 0 to numBits(start, end) - 1     // Traverse the bitmap
        if b % BITS_IN_BLOCK = 0              // If crossed to new block
            offset[block] <- loc              // Update the offset of this block to be loc
            block <- block + 1                // Advance the block
        if bitmap[b] = MARKED                 // Advance loc for every marked bit
            loc <- loc + BYTES_PER_BIT
```

updateReferencesRelocate

```
updateReferencesRelocate(start, end):
    for each fld in Roots                           // Assign new addresses for the Roots
        ref <- *fld
        if ref != null
            *fld <- newAddress(ref)

    scan <- start
    while scan < end
        scan <- nextMarkedObject(scan)              // Move until the next object with a set bit
        for each fld in Pointers(scan)              // Find new addresses for its every child
            ref <- *fld
            if ref != null
                *fld <- newAddress(ref)

        dest <- newAddress(scan)                    // Find a new address for the object itself
        move(scan, dest)                            // Make the actual move
```