Do It Yourself (OpenJDK) Garbage Collector

Adding garbage collection to our non-garbage collector: wait, what?

Table of Contents

Aleksey Shipilëv, JVM/Performance Geek, redhat.
Shout out at Twitter: @shipilev
Questions, comments, suggestions: aleksey@shipilev.net

> This post is also available in ePUB and mobi.

Thanks to Richard Startin, Roman Kennke, Cyrille Chépélov and others for reviews, edits and helpful suggestions!

Introduction

Building any part of language runtime is a fun exercise. At least, building the first and hacky version is! Building a reliable, high-performance, observable, debuggable, fail-predictably version of the runtime subsystem is really, really hard.

Building a simple garbage collector is deceptively easy, and this is what we shall do here. Roman Kennke did the FOSDEM 2019 talk and demo "Building the GC, in 20 minutes" using an earlier revision of this patch. While the code in the actual implementation is rich with in-place comments, some sort of over-arching narrative around it is in order, hence this post.

Basic understanding how garbage collectors work would improve the reading experience a lot. There is a bit of discussion here and there about general ideas and Hotspot specifics, but it is not really a place for a crash course in GC construction. Pick up the GC Handbook and read its first chapters on basic GCs, or get up to speed with Wikipedia article.

1. Building Blocks

Implementing a new GC when many other GCs are already implemented is much simpler, because there are a lot of existing building blocks that can be (re)used to offload parts of implementation to proven and tested code.

1.1. Epsilon GC

JEP 318: "Epsilon: A No-Op Garbage Collector (Experimental)"was introduced in OpenJDK 11. The goal for that implementation is to provide the minimal implementation that can be used when memory reclamation is not required or indeed is prohibited. Read the JEP for more discussion when it can be useful.

From the implementation perspective, "garbage collector" is a misnomer term, the proper term would be "automatic memory manager", which takes care of both allocation and reclamation of memory. As Epsilon GC implements only the "allocation" part, having no opinion how should the "reclamation" part look like, it is a good blank slate to implement the real GC algorithm on top of it.

1.1.1. Allocation

The most developed part of Epsilon GC is allocation path. It services the external requests to allocate memory of arbitrary size, or allocating the Thread-Local Allocation Buffer (TLAB) of given size. The implementation itself tries not to expand TLABs too much, because as there is no reclamation going on, bytes wasted are bytes never reclaimed.

1.1.2. Barriers

Some garbage collectors require interaction with the application to maintain GC invariants, by forcing runtime and application to perform *GC barriers* when accessing the heap. This is universally true for all concurrent collectors, and is true for many generational stop-the-world collectors.

Epsilon does not need barriers, but runtime and compilers still need to *know* that barriers are no-ops. Hooking that up everywhere can be tedious. Fortunately, since OpenJDK 11, there is JEP-304: "Garbage Collection Interface" that makes inserting barriers much, much, much cleaner. Notably, Epsilon's barrier set is empty, delegating all trivial work — actual loads, stores, CASes, arraycopies — to the basic barrier set. If we are to build a GC that still doesn't require barriers, we can simply reuse what Epsilon already has.

1.1.3. Monitoring Hooks

The last tedious part of implementing the GC is hooking it up to various monitoring facilities inside the JVM: MX beans have to work, diagnostic commands have to work, etc. Epsilon already handles this for us.

1.2. Runtime and GC

1.2.1. Roots

Garbage collectors generally need to know which parts of Java runtime hold references to Java heap. These locations, *GC Roots*, include thread stacks and local variables (including the ones in JIT-compiled code!), native classes and classloaders, JNI handles, etc. Knowing what the root set is can be daunting. But in Hotspot, those locations are tracked by each of the VM subsystems, and we can carefully study what existing GC implementations do with them. You would see that in the implementation section below.

1.2.2. Object Walkers

Garbage collectors also need to walk outbound references from Java objects. Since this is a ubiquitous operation, shared runtime parts already provide GCs with object walkers that GCs can use, obviating the need for us to implement it ourselves. You would see the calls to, for example, `obj→oop_iterate` in the implementation section below.

1.2.3. Forwarding Data

Moving garbage collectors need to record the new addresses for moved objects somewhere. There are several places where to store that *forwarding data*:

1. Reuse the "mark word" in the object(Serial, Parallel, modern Shenandoah,[1] etc). When the world is stopped, all accesses to object words are controlled, and no Java thread would see whatever transient data we store in the mark word. We can reuse this word for storing forwarding data.
2. Maintain separate native forwarding table (ZGC, C4, others). This would completely isolate GC from runtime and the rest of the application, as only GC would know such forwarding table exists. This is why concurrent GCs usually employ this scheme: they want to avoid messing with other things.

3. Add another word to the object (legacy Shenandoah, others). This would combine both approaches, by letting runtime and application work with existing headers without problems, while still keeping forwarding data around.

1.2.4. Marking Data

Garbage collectors need to record the reachability (marking) data somewhere. Again, there are several ways where to store it:

1. Reuse the "mark word" in the object (Serial, Parallel, etc). Again, in stop-the-world mode, we can use the bits in that mark word to encode the "marked" property. Then, if we need to walk all live objects, we walk the heap object by object, exploiting heap parsability.
2. Maintain a separate marking data structure (G1, Shenandoah, etc). This usually comes as the separate bitmap that maps every N bytes of the Java heap to 1 bit of marking bitmap. Normally, Java objects are aligned by 8 bytes, so the marking bitmap maps 64 bits of heap to 1 bit of marking bitmap, taking 1/64-th of heap size in native memory. This overhead that is paid off handsomely when scanning the heap for live objects, especially sparse ones: walking the bitmap is usually much faster than walking the parsable heap object per object.
3. Encode marking information in the references themselves (ZGC, C4, others). This requires coordination with application to strip the marking information from the reference, or do other tricks to maintain correctness. In other words, it requires GC barriers or more GC work.

2. Grand Plan

Arguably the easiest GC to implement on top of Epsilon is LISP2-style Mark-Compact. The basic idea for this GC is given in a relevant Wikipedia entry, or in the GC Handbook, Chapter 3.2. There are sketches of what the algorithm does through the implementation section below, but you are encouraged to read the Wikipedia bit or GC Handbook to understand what we are about to do.

The algorithm in question is *sliding* GC: it moves objects by sliding them to the start of the heap. It comes with the following characteristics that have upsides and downsides:

- It maintains the allocation order. This is very good for controlling memory layout if you are into that kind of thing (control freaks, rejoice!), but on the down side, you don’t get the automatic locality of reference(booo!).
- It is O(N) for the number of objects. However, that linearity comes at a cost, requiring the GC to walk over heap 4 times per GC cycle.
- It does not need any free Java heap memory! There no need to reserve heap memory to evacuate live objects to, so even a 99.(9)% full heap can be operated on. If we go with other ideas for a simple GC, for example semi-space scavenger, we would need to rewire how heap is represented a little bit, plus reserve some space

for evacuation, which goes beyond the scope of this exercise.

- With some engineering, it can yield zero footprint and time overhead when GC is not active. It starts with whatever state the allocation area is in, and leaves with a densely compacted one. This fits Epsilon representation very well: it would just continue allocating from the compaction point. This is also its downside: a few dead objects at the beginning of the heap induce lots of moves.
- It does not require any new barriers, which keeps `EpsilonBarrierSet` untouched.

For simplicity, the GC implementation would be fully stop-the-world, non-generational, and single-threaded. In this case, it makes sense to use a marking bitmap to store marking data, and reuse mark words for storing forwarding data.

3. Implementing GC Core

As it might be too daunting to read the entire implementation, this section tries to introduce it piece by piece.

3.1. Prologue

GC usually needs to do a few things in preparation for GC. Read the comments, they should be self-explanatory:

```
{
  GCTraceTime(Info, gc) time("Step 0: Prologue", NULL);

  // Commit marking bitmap memory. There are several upsides of doing this
  // before the cycle: no memory is taken if GC is not happening, the memory
  // is "cleared" on first touch, and untouched parts of bitmap are mapped
  // to zero page, boosting performance on sparse heaps.
  if (!os::commit_memory((char*)_bitmap_region.start(), _bitmap_region.byte_size(), false)) {
    log_warning(gc)("Could not commit native memory for marking bitmap, GC failed");
    return;
  }

  // We do not need parsable heap for this algorithm to work, but we want
  // threads to give up their TLABs.
  ensure_parsability(true);

  // Tell various parts of runtime we are doing GC.
  CodeCache::gc_prologue();
  BiasedLocking::preserve_marks();

  // Derived pointers would be re-discovered during the mark.
  // Clear and activate the table for them.
  DerivedPointerTable::clear();
}
```

Since we are going to use a marking bitmap to track what objects are reachable, we need to clean it up before using it. Or, in this case, as we pursue the goal of never taking resources until the GC cycle hits, we need to commit the bitmap to memory first. This comes with a few interesting advantages, at least on Linux, where most of the bitmap would be mapped to zero page, especially for sparse heap.

Threads need to give up their TLABs and ask GC for new ones after GC finishes. [2]

Some parts of runtime, especially those parts that deal with references to Java heap, get broken by GC, so we need to notify them that GC is about to act. This would let those subsystems to prepare/save parts of their state before impending GC moves.

3.2. Marking

Stop-the-world marking is quite simple once we have all the pieces ready to go. Marking is pretty generic, and it would normally be the first step in many GC implementations.

```
{
  GCTraceTime(Info, gc) time("Step 1: Mark", NULL);

  // Marking stack and the closure that does most of the work. The closure
  // would scan the outgoing references, mark them, and push newly-marked
  // objects to stack for further processing.
  EpsilonMarkStack stack;
  EpsilonScanOopClosure cl(&stack, &_bitmap);

  // Seed the marking with roots.
  process_roots(&cl);
  stat_reachable_roots = stack.size();

  // Scan the rest of the heap until we run out of objects. Termination is
  // guaranteed, because all reachable objects would be marked eventually.
  while (!stack.is_empty()) {
    oop obj = stack.pop();
    obj->oop_iterate(&cl);
    stat_reachable_heap++;
  }

  // No more derived pointers discovered after marking is done.
  DerivedPointerTable::set_active(false);
}
```

It is like every other graph walking problem: you start from the initial set of reachable vertices, walk all outbound edges, recording which vertices you visited, and do this until you run out of unvisited verticies. In GC, "vertexes" are the objects, and "edges" are the references between them.

Technically, we could have just used recursion to walk the object graph, but it is a bad idea for arbitrary graphs, which can have very large diameters. Think about walking the linked list with 1 billion nodes in it. So, to limit the recursion depth, we use marking stack that records objects discovered.

The initial set of reachable objects comes from GC roots. Don't dwell on what `process_roots` does for now, we will visit it later. So far, assume it walks all references reachable from the VM side.

The marking bitmap serves both as the thing that tracks the *marking wavefront* (the set of already visited objects), and in the end gives us the desired output: the set of all reachable objects. The actual work in done in the `EpsilonScanOopClosure`,[3] that would be applied to all interesting objects, and which iterates all references in a given object, and here it is:

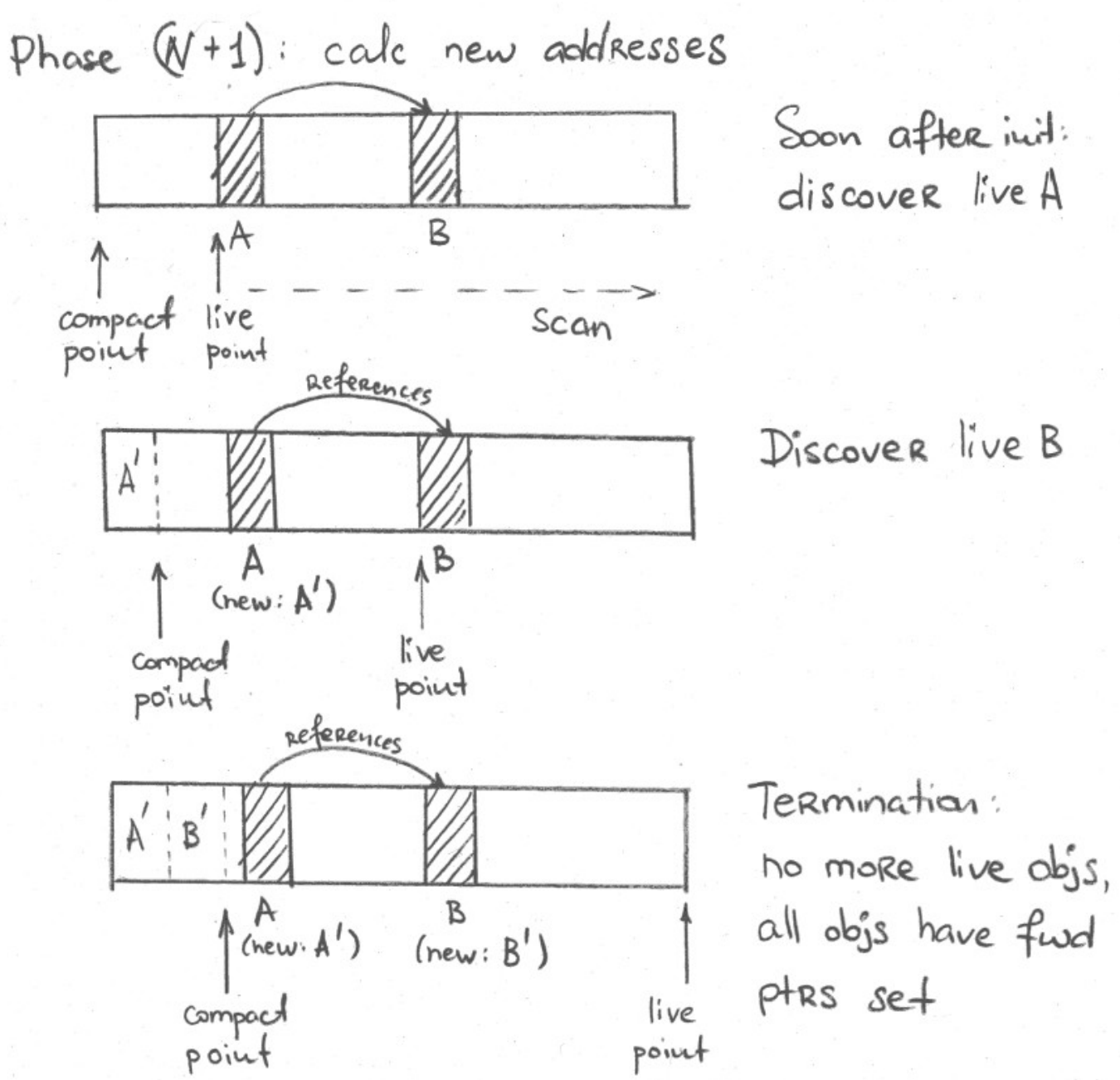
```
class EpsilonScanOopClosure : public BasicOopIterateClosure {
private:
  EpsilonMarkStack* const _stack;
  MarkBitMap* const _bitmap;

  template <class T>
  void do_oop_work(T* p) {
    // p is the pointer to memory location where oop is, load the value
    // from it, unpack the compressed reference, if needed:
    T o = RawAccess<>::oop_load(p);
    if (!CompressedOops::is_null(o)) {
      oop obj = CompressedOops::decode_not_null(o);

      // Object is discovered. See if it is marked already. If not,
      // mark and push it on mark stack for further traversal. Non-atomic
      // check and set would do, as this closure is called by single thread.
      if (!_bitmap->is_marked(obj)) {
        _bitmap->mark((HeapWord*)obj);
        _stack->push(obj);
      }
    }
  }
};
```

After this step is done, `_bitmap` contains the bits set at the locations where alive objects are. This gives us the opportunity to walk all live objects, like this:

```
// Walk the marking bitmap and call object closure on every marked object.
// This is much faster that walking a (very sparse) parsable heap, but it
// takes up to 1/64-th of heap size for the bitmap.
void EpsilonHeap::walk_bitmap(ObjectClosure* cl) {
   HeapWord* limit = _space->top();
   HeapWord* addr = _bitmap.get_next_marked_addr(_space->bottom(), limit);
   while (addr < limit) {
     oop obj = oop(addr);
     assert(_bitmap.is_marked(obj), "sanity");
     cl->do_object(obj);
     addr += 1;
     if (addr < limit) {
       addr = _bitmap.get_next_marked_addr(addr, limit);
     }
   }
}
```

3.3. Calculating New Locations

This part is also quite simple, and it implements what the algorithm says.


Phase (N+1): calc new addresses
A
B
compact point
live point
Scan
Soon after init: discover live A
References
A'
A
(new: A')
B
compact point
live point
Discover live B
References
A'
B'
A
(new: A')
B
(new: B')
compact point
live point
Termination: no more live objs, all objs have fwd ptrs set

```
  // We are going to store forwarding information (where the new copy resides)
  // in mark words. Some of those mark words need to be carefully preserved.
  // This is an utility that maintains the list of those special mark words.
  PreservedMarks preserved_marks;

  // New top of the allocated space.
  HeapWord* new_top;

  {
    GCTraceTime(Info, gc) time("Step 2: Calculate new locations", NULL);

    // Walk all alive objects, compute their new addresses and store those
    // addresses in mark words. Optionally preserve some marks.
    EpsilonCalcNewLocationObjectClosure cl(_space->bottom(), &preserved_marks);
    walk_bitmap(&cl);

    // After addresses are calculated, we know the new top for the allocated
    // space. We cannot set it just yet, because some asserts check that objects
    // are "in heap" based on current "top".
    new_top = cl.compact_point();

    stat_preserved_marks = preserved_marks.size();
  }
```

The only wrinkle here is that we store the new addresses in the mark words of the Java objects, and these mark words could be busy with something else, for example, locking information. Luckily, those non-trivial mark words are rare, and we can just store them separately if needed: that is what `PreservedMarks` would do for us.

The actual `EpsilonCalcNewLocationObjectClosure` does what the algorithm step wants:

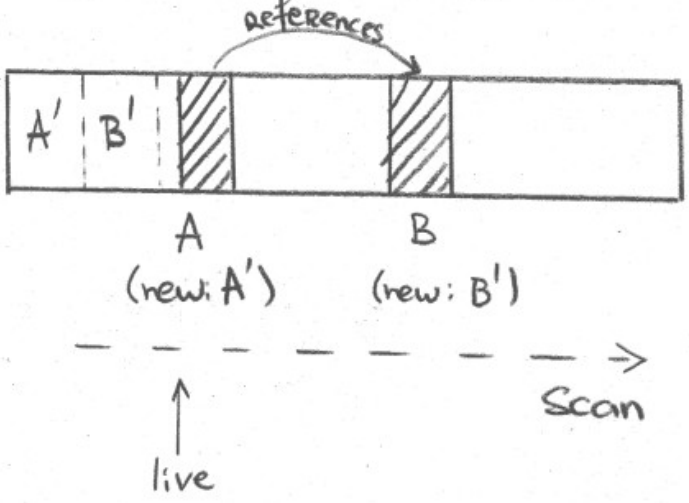
```
class EpsilonCalcNewLocationObjectClosure : public ObjectClosure {
private:
  HeapWord* _compact_point;
  PreservedMarks* const _preserved_marks;

public:
  EpsilonCalcNewLocationObjectClosure(HeapWord* start, PreservedMarks* pm) :
                                      _compact_point(start),
                                      _preserved_marks(pm) {}

  void do_object(oop obj) {
    // Record the new location of the object: it is current compaction point.
    // If object stays at the same location (which is true for objects in
    // dense prefix, that we would normally get), do not bother recording the
    // move, letting downstream code ignore it.
    if ((HeapWord*)obj != _compact_point) {
      markOop mark = obj->mark_raw();
      if (mark->must_be_preserved(obj)) {
        _preserved_marks->push(obj, mark);
      }
      obj->forward_to(oop(_compact_point));
    }
    _compact_point += obj->size();
  }

  HeapWord* compact_point() {
    return _compact_point;
  }
};
```

`forward_to` is the critical part here: it stores the "forwarding address" in the object's mark word. We are going to need it in the later steps.

3.4. Adjusting The Pointers

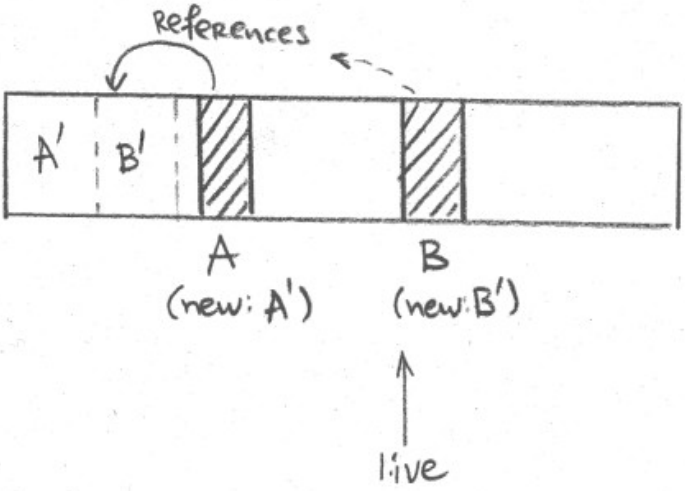
Walk the heap again and rewrite all references to their new locations, as per algorithm:

Phase (N+2): Adjusting Refs



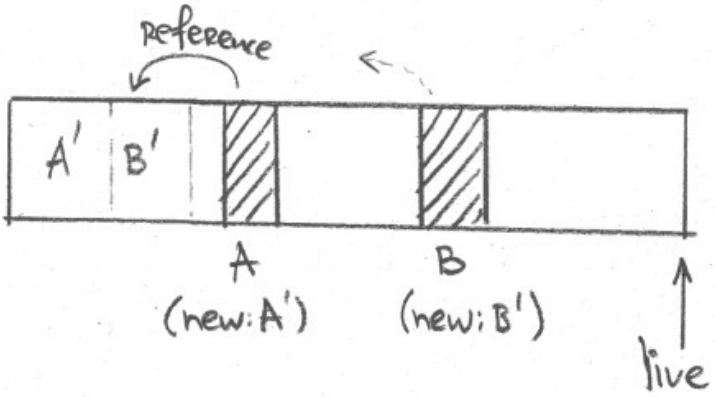
Initial: References to old locations

Scan all live objs



Step: Rewrite refs to new locations

Note: heap is pretty much broken now: Refs point to „nowhere"



Termination: no more live objs

```
{
  GCTraceTime(Info, gc) time("Step 3: Adjust pointers", NULL);

  // Walk all alive objects _and their reference fields_, and put "new
  // addresses" there. We know the new addresses from the forwarding data
  // in mark words. Take care of the heap objects first.
  EpsilonAdjustPointersObjectClosure cl;
  walk_bitmap(&cl);

  // Now do the same, but for all VM roots, which reference the objects on
  // their own: their references should also be updated.
  EpsilonAdjustPointersOopClosure cli;
  process_roots(&cli);

  // Finally, make sure preserved marks know the objects are about to move.
  preserved_marks.adjust_during_full_gc();
}
```

There are two sets of references to our moved objects; from other objects in the heap itself and from the GC roots. We need to update both. Some preserved marks have recorded the object references too, so we need to ask them to update themselves. `PreservedMarks` knows what to do, because it expects "forwarding data" in the same location we have recorded it at, in object mark word.

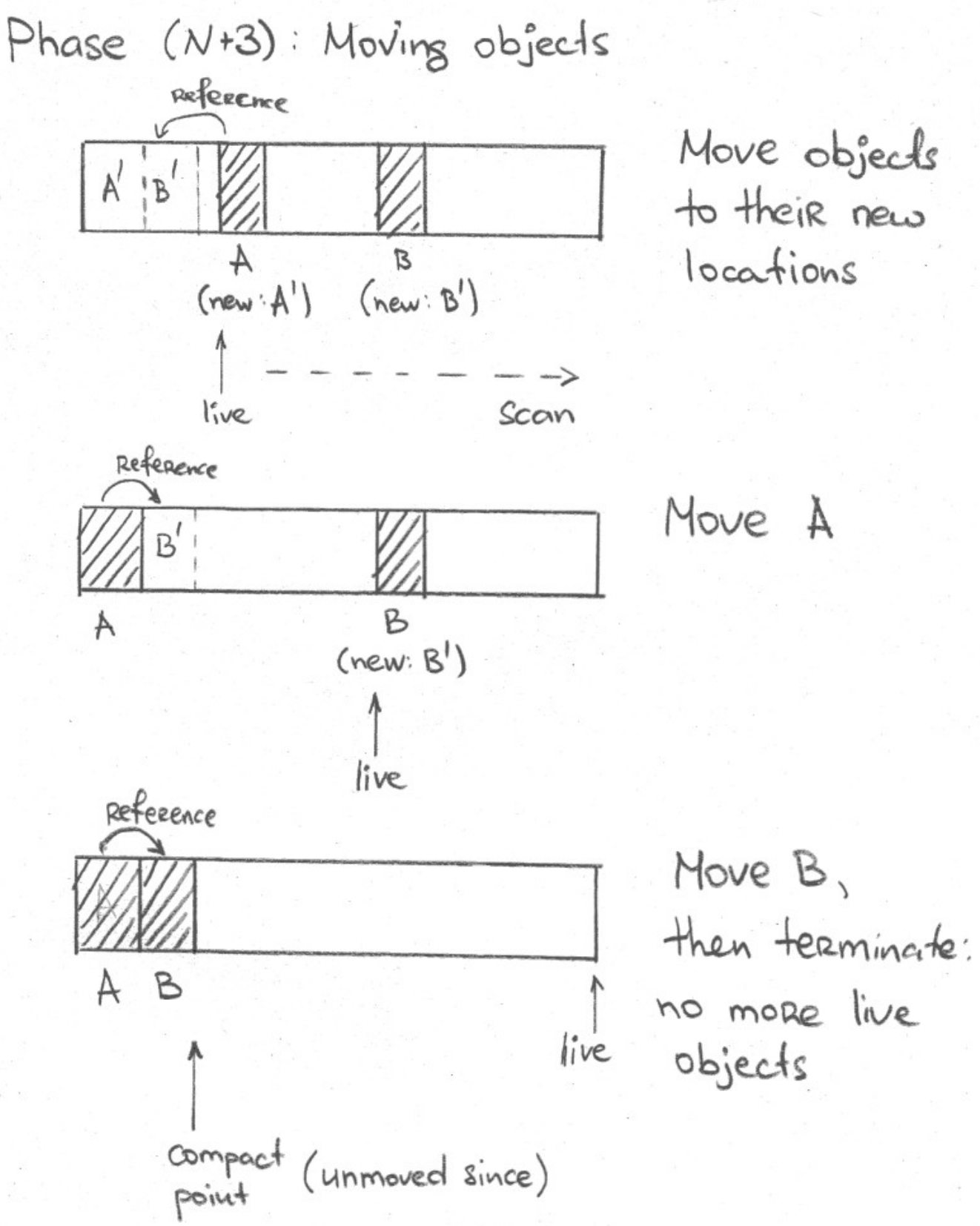
The closures are now coming in two types: one that accepts the objects and walks its contents, and the other one that updates the locations. Here is a little performance optimization: if an object is not forwarded, it does not move, so we can spare quite a few heap writes:

```
class EpsilonAdjustPointersOopClosure : public BasicOopIterateClosure {
private:
  template <class T>
  void do_oop_work(T* p) {
    // p is the pointer to memory location where oop is, load the value
    // from it, unpack the compressed reference, if needed:
    T o = RawAccess<>::oop_load(p);
    if (!CompressedOops::is_null(o)) {
      oop obj = CompressedOops::decode_not_null(o);

      // Rewrite the current pointer to the object with its forwardee.
      // Skip the write if update is not needed.
      if (obj->is_forwarded()) {
        oop fwd = obj->forwardee();
        assert(fwd != NULL, "just checking");
        RawAccess<>::oop_store(p, fwd);
      }
    }
  }
};

class EpsilonAdjustPointersObjectClosure : public ObjectClosure {
private:
  EpsilonAdjustPointersOopClosure _cl;
public:
  void do_object(oop obj) {
    // Apply the updates to all references reachable from current object:
    obj->oop_iterate(&_cl);
  }
};
```

After this step is done, the heap is basically corrupted: references point to "wrong" locations, where objects have not been moved yet. Let’s rectify that!

3.5. Moving The Objects

Time to move the objects into their new locations, as per algorithm step:



Walk the heap again and apply the `EpsilonMoveObjectsObjectClosure` closure to all live objects:

```
{
  GCTraceTime(Info, gc) time("Step 4: Move objects", NULL);

  // Move all alive objects to their new locations. All the references are
  // already adjusted at previous step.
  EpsilonMoveObjectsObjectClosure cl;
  walk_bitmap(&cl);
  stat_moved = cl.moved();

  // Now we moved all objects to their relevant locations, we can retract
  // the "top" of the allocation space to the end of the compacted prefix.
  _space->set_top(new_top);
}
```

After this is done, we can retract the allocated space to the compaction point, letting the allocation path allocate from there after GC cycle is over.

Note that sliding GC means we can *overwrite* the contents of existing objects, but since we are scanning in one direction, that means the object we are overwriting is already copied out to its proper location. [4] So, the closure itself just moves the forwarded objects to their new locations:

```
class EpsilonMoveObjectsObjectClosure : public ObjectClosure {
public:
  void do_object(oop obj) {
    // Copy the object to its new location, if needed. This is final step,
    // so we have to re-initialize its new mark word, dropping the forwardee
    // data from it.
    if (obj->is_forwarded()) {
      oop fwd = obj->forwardee();
      assert(fwd != NULL, "just checking");
      Copy::aligned_conjoint_words((HeapWord*)obj, (HeapWord*)fwd, obj->size());
      fwd->init_mark_raw();
    }
  }
};
```

3.6. Epilogue

GC is over, the heap is now almost consistent again, we just need a few finishing touches:

```
{
  GCTraceTime(Info, gc) time("Step 5: Epilogue", NULL);

  // Restore all special mark words.
  preserved_marks.restore();

  // Tell the rest of runtime we have finished the GC.
  DerivedPointerTable::update_pointers();
  BiasedLocking::restore_marks();
  CodeCache::gc_epilogue();
  JvmtiExport::gc_epilogue();

  // Marking bitmap is not needed anymore
  if (!os::uncommit_memory((char*)_bitmap_region.start(), _bitmap_region.byte_size())) {
    log_warning(gc)("Could not uncommit native memory for marking bitmap");
  }

  // Return all memory back if so requested. On large heaps, this would
  // take a while.
  if (EpsilonUncommit) {
    _virtual_space.shrink_by((_space->end() - new_top) * HeapWordSize);
    _space->set_end((HeapWord*)_virtual_space.high());
  }
}
```

Notify other parts of runtime to make their post-GC cleanups/fixups. Restore the special mark words we saved before. Kiss the marking bitmap bye-bye, we do not need it anymore.

And, if we are so inclined, we can retract the committed space to the new allocation point, thus returning memory to the OS!

4. Hooking GC To VM

4.1. Roots Walking

Remember the need to walk the special implicitly-reachable references from the VM? This is achieved by asking every special subsystem in VM to walk the references it has hidden from the rest of Java objects. The exhaustive list of roots in current Hotspot looks something like this:

```
void EpsilonHeap::do_roots(OopClosure* cl) {
  // Need to tell runtime we are about to walk the roots with 1 thread
  StrongRootsScope scope(1);

  // Need to adapt oop closure for some special root types.
  CLDToOopClosure clds(cl, ClassLoaderData::_claim_none);
  MarkingCodeBlobClosure blobs(cl, CodeBlobToOopClosure::FixRelocations);

  // Walk all these different parts of runtime roots. Some roots require
  // holding the lock when walking them.
  {
    MutexLockerEx lock(CodeCache_lock, Mutex::_no_safepoint_check_flag);
    CodeCache::blobs_do(&blobs);
  }

  ClassLoaderDataGraph::cld_do(&clds);
  Universe::oops_do(cl);
  Management::oops_do(cl);
  JvmtiExport::oops_do(cl);
  JNIHandles::oops_do(cl);
  WeakProcessor::oops_do(cl);
  ObjectSynchronizer::oops_do(cl);
  SystemDictionary::oops_do(cl);
  Threads::possibly_parallel_oops_do(false, cl, &blobs);
}
```

There are extensions that walk the roots concurrently and/or in parallel. For our single-threaded GC case, simple walks are enough.

4.2. Safepoints and Stop-The-World

Since our GC is stop-the-world, we need to request VM to perform the actual stop-the-world pause. In Hotspot, this is achieved by implementing a brand new `VM_Operation` that calls into our GC code, and asking VM thread to execute it:

```
// VM operation that executes collection cycle under safepoint
class VM_EpsilonCollect: public VM_Operation {
private:
  const GCCause::Cause _cause;
  EpsilonHeap* const _heap;
  static size_t _last_used;
public:
  VM_EpsilonCollect(GCCause::Cause cause) : VM_Operation(),
                                            _cause(cause),
                                            _heap(EpsilonHeap::heap()) {};

  VM_Operation::VMOp_Type type() const { return VMOp_EpsilonCollect; }
  const char* name()             const { return "Epsilon Collection"; }

  virtual bool doit_prologue() {
    // Need to take the Heap lock before managing backing storage.
    // This also naturally serializes GC requests, and allows us to coalesce
    // back-to-back allocation failure requests from many threads. There is no
    // need to handle allocation failure that comes without allocations since
    // last complete GC. Waiting for 1% of heap allocated before starting next
    // GC seems to resolve most races.
    Heap_lock->lock();
    size_t used = _heap->used();
    size_t capacity = _heap->capacity();
    size_t allocated = used > _last_used ? used - _last_used : 0;
    if (_cause != GCCause::_allocation_failure || allocated > capacity / 100) {
      return true;
    } else {
      Heap_lock->unlock();
      return false;
    }
  }

  virtual void doit() {
    _heap->entry_collect(_cause);
  }

  virtual void doit_epilogue() {
    _last_used = _heap->used();
    Heap_lock->unlock();
  }
};

size_t VM_EpsilonCollect::_last_used = 0;

void EpsilonHeap::vmentry_collect(GCCause::Cause cause) {
  VM_EpsilonCollect vmop(cause);
  VMThread::execute(&vmop);
}
```

This also helps solving a few performance-sensitive races when all threads want to perform GC at once, as it usually happens when memory is exhausted.

4.3. Allocation Failures

While entering the GC on explicit request is good, we also want GC to react on heap exaustion, when there is no memory left. It is enough to replace most `allocate_work` calls with this handy wrapper that does GC on allocation failure:

```
HeapWord* EpsilonHeap::allocate_or_collect_work(size_t size) {
  HeapWord* res = allocate_work(size);
  if (res == NULL && EpsilonSlidingGC) {
    vmentry_collect(GCCause::_allocation_failure);
    res = allocate_work(size);
  }
  return res;
}
```

There! All done.

5. Building

The patch should apply over head OpenJDK without problems.

```
$ hg clone https://hg.openjdk.java.net/jdk/jdk/ jdk-jdk
$ cd jdk-jdk
$ curl https://shipilev.net/jvm/diy-gc/webrev/jdk-jdk-epsilon.changeset | patch -p1
```

Then, build OpenJDK as usual:

```
$ ./configure --with-debug-level=fastdebug
$ make images
```

Then, run it as usual:

```
$ build/linux-x86_64-server-fastdebug/images/jdk/bin/java -XX:+UnlockExperimentalVMOptions -
XX:+UseEpsilonGC -XX:+EpsilonSlidingGC -version
openjdk version "13-internal" 2019-09-17
OpenJDK Runtime Environment (build 13-internal+0-adhoc.shade.jdk-jdk-epsilon)
OpenJDK 64-Bit Server VM (build 13-internal+0-adhoc.shade.jdk-jdk-epsilon, mixed mode, sharing)
```

6. Testing

How do you make sure the GC implementation is not broken? Well, there are a few things that are usually handy:

1. Asserts. Lots of asserts. Hotspot code does assert a lot, so running with *fastdebug* builds usually shows up lots of interesting failures here and there when GC is broken.
2. Internal verification. Current patch implements the final step in GC cycle that walks all live objects and verifies they are sane. This usually catches egregious errors before they are exposed to runtime and application when GC cycle is over.
3. Tests. Asserts and verification is useless if code does not actually run. Having unit- and integration-sized tests that can be run early and often is very handy.

For example, this is how you can verify the patch is not horribly broken:

```
$ CONF=linux-x86_64-server-fastdebug make images run-test TEST=gc/epsilon/
Building targets 'images run-test' in configuration 'linux-x86_64-server-fastdebug'
Test selection 'gc/epsilon/', will run:
* jtreg:test/hotspot/jtreg/gc/epsilon

Running test 'jtreg:test/hotspot/jtreg/gc/epsilon'
Passed: gc/epsilon/TestAlwaysPretouch.java
Passed: gc/epsilon/TestAlignment.java
Passed: gc/epsilon/TestElasticTLAB.java
Passed: gc/epsilon/TestEpsilonEnabled.java
Passed: gc/epsilon/TestHelloWorld.java
Passed: gc/epsilon/TestLogTrace.java
Passed: gc/epsilon/TestDieDefault.java
Passed: gc/epsilon/TestDieWithOnError.java
Passed: gc/epsilon/TestMemoryPools.java
Passed: gc/epsilon/TestMaxTLAB.java
Passed: gc/epsilon/TestPrintHeapSteps.java
Passed: gc/epsilon/TestArraycopyCheckcast.java
Passed: gc/epsilon/TestClasses.java
Passed: gc/epsilon/TestUpdateCountersSteps.java
Passed: gc/epsilon/TestDieWithHeapDump.java
Passed: gc/epsilon/TestByteArrays.java
Passed: gc/epsilon/TestManyThreads.java
Passed: gc/epsilon/TestRefArrays.java
Passed: gc/epsilon/TestObjects.java
Passed: gc/epsilon/TestElasticTLABDecay.java
Passed: gc/epsilon/TestSlidingGC.java
Test results: passed: 21
TEST SUCCESS
```

Satisfied with this? Now run the actual application with `fastdebug` build and verfication enabled. Does not crash? One must be hopeful at that point.

7. Performance

Let's take spring-petclinic, load it up with Apache Bench, run it with our toy GC! As the workload has little live data, both generational and non-generational GCs should be happy with it.

Run with `-Xlog:gc -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -XX:+EpsilonSlidingGC` :

```
Heap: 20480M reserved, 20480M (100.00%) committed, 19497M (95.20%) used
GC(2) Step 0: Prologue 2.085ms
GC(2) Step 1: Mark 51.005ms
GC(2) Step 2: Calculate new locations 71.207ms
GC(2) Step 3: Adjust pointers 49.671ms
GC(2) Step 4: Move objects 22.839ms
GC(2) Step 5: Epilogue 1.008ms
GC(2) GC Stats: 70561 (8.63%) reachable from roots, 746676 (91.37%) reachable from heap, 91055
(11.14%) moved, 2237 (0.27%) markwords preserved
GC(2) Heap: 20480M reserved, 20480M (100.00%) committed, 37056K (0.18%) used
GC(2) Lisp2-style Mark-Compact (Allocation Failure) 20479M->36M(20480M) 197.940ms
```

200 ms? Not bad for a first-try single-threaded GC we just built! You can see that the four major phases take the same order of magnitude time. In fact, if you play around with different heap occupancies and heap sizes, then the pattern would emerge: more live data means significantly slower GC (touching all live objects is not fun when there are many of them), more heap size means a bit slower GC (walking long distances even over sparse heap takes a toll on throughput).

For comparison, generational GC and scavengers have a field day with this workload. For example, `-Xlog:gc -XX:+UseSerialGC` which does mostly young collections:

```
GC(46) Pause Young (Allocation Failure) 575M->39M(1943M) 2.603ms
GC(47) Pause Young (Allocation Failure) 575M->39M(1943M) 2.606ms
GC(48) Pause Young (Allocation Failure) 575M->39M(1943M) 2.747ms
GC(49) Pause Young (Allocation Failure) 575M->39M(1943M) 2.578ms
```

Whoa. 2 ms. This is because most objects are dead in young generation, and there is hardly any GC work to do. If we disable generational extensions from `-Xlog:gc -XX:+UseSerialGC` and force Full GCs only, then we would see much less rosy picture:

```
GC(3) Pause Full (Allocation Failure) 16385M->34M(18432M) 1969.694ms
GC(4) Pause Full (Allocation Failure) 16385M->34M(18432M) 2261.405ms
GC(5) Pause Full (Allocation Failure) 16385M->34M(18432M) 2327.577ms
GC(6) Pause Full (Allocation Failure) 16385M->34M(18432M) 2328.976ms
```

There are plenty of other metrics and scenarios one can play with. This is left as an exercise for the reader.

8. Where To Go From Here

There are many things the implementation can go from here. It would encroach on what existing OpenJDK GCs are already doing (and tested for!) though, so it should be taken as educational exercise only.

What can be improved:

1. Implement reference processing. The current implementation ignores the fact soft/weak/phantom references exist. It also ignores the existence of finalizeable objects. This is not ideal from performance standpoint, but it is safe from the correctness POV, because shared code "just" treats all those references as always reachable,[5] so would get moved and updated as any other regular reference. A proper implementation would involve hooking up shared `ReferenceProcessor` to marking code, and marking/clearing the surviving/dead references after mark is over.
2. Implement class unloading and other VM cleanups. The current implementation never unloads the classes, nor does it clear up internal VM datastructures that hold the objects unreachable from heap, and therefore are likely redundant. Implementing this would require taking care of *weak* and *strong* roots, marking only *strong* roots by default, and then seeing if any of the *weak* roots are still marked after mark is finished, cleaning up the dead ones.
3. Parallelize it.[6] The simplest way to make the parallel version is to divide the heap into the per-thread regions, and do the same sequential compaction within those regions. This would leave the gaps between the regions, so the allocation path needs to get modified to know there are more than one free region.
4. Implement dense prefix handling.[7] It is usually the case that normal heaps end up with "sediment" layers of always reachable objects, and we can save quite a few cycles if we designate some prefix of the heap as non-moving. Then, we can avoid calculating addresses and moving objects within the dense prefix. With some tricks, we can also avoid adjusting pointers in dense prefix, if we are able to detect interesting objects (those having references outside the prefix) in dense prefix during the mark. We would still need to mark through prefix anyway.
5. Extend dense prefix to full-blown generational. With some GC barrier work, we can tell what parts of dense prefix are interesting to look at *before doing the mark*, thus saving time marking through it. In the end, it would turn out to be "generational", which would do "young" collections past the prefix, and sometimes do "full" collections that would compact everything, including the prefix.
6. Take whatever GC algorithm from the GC Handbook, and try to implement it.

Conclusion

What to conclude here? Implementing toy GCs are fun, educational, and probably a good fit for an university GC course.

Productizing such the implementation would be a tedious and time-consuming process, so it is easier to switch to and tune existing collectors. Notably, if developed further, this implementation would eventually morph into already existing Serial or Parallel GC, rendering the development efforts more or less futile.

---

1. This changed with elimination of separate forwarding pointer word. Modern Shenandoah implementation uses the mark word in "old" object to store the forwarding data, and barriers make sure that copy is never exposed to application.

2. Do not confuse TLABs and `java.lang.ThreadLocal`. From the GC perspective, ThreadLocals are still pretty regular objects, and they would not be cleared by GC, unless specially instructed from Java code

3. See, Java used closures before it was cool!

4. The new and old location for the given object can overlap though. For example, you can slide a 100 byte object by 8 bytes. The copy routine would make sure the overlapping contents are copied correctly, notice `Copy::aligned_*conjoint*_words`.

5. From the GC perspective, `java.lang.ref.Reference.referent` is just another Java field, which would be strongly reachable, unless we walk the heap in a special way. Finalizable objects have their own synthetic instances of `FinalReference` that hold on to them.

6. The parallel versions of this mark-compact are implemented as Full GC fallbacks in Shenandoah (starting with OpenJDK 8) and G1 (starting from OpenJDK 10, after JEP 307: "Parallel Full GC for G1")

7. For fun, I made the follow-up patch that seems to do it. It is not very well tested, and makes a few assumptions about heap structure along the way.