ClassUnloading

A discussion on how to implement class unloading in DRLVM is currently being discussed on the mailing list. The ideas from the discussion has been collected on this page.

Refresh on the problem statement

The classloader can be unloaded together with all classes defined by it if the following conditions are met

- classloader is unreachable
- there are no activation records (method stack frames) of the methods of classes defined by the classloader
- there are no instances of the classes defined by the classloader

The most difficult problem to solve efficiently is to determine if instances of the classes exist.

Design ideas

Determine absence of class instances

- Allocate a mark bit for each `VTable`, unconditionally write 1 when scanning object (Robin)
  - *Issue*: false sharing if mark bit is in the same cache line as gcmap. Relevant only for parallel GC algorithms on a true multiprocessor with separate cache
- Can be addressed by using a 'side byte-map' for these mark bytes.
- Trace VTables as regular java objects, add a reference from `VTable` to `java.lang.Class` (Alexey)
  - *Issue*: `VTable` objects require pinned allocation
  - *Issue*: amount of tracing work increases for each garbage collection

Determine absence of active stack frames

- Iterate over thread stacks and mark `VTables` of active stack frames (Etienne)

End-to-end design proposals

"Automatic VTable" design (Alexey)

"Automatic class unloading" means that j.l.Classloader instance is unloaded automatically (w/o additional enumeration tricks or GC dependency) and after we detect that some class loader was unloaded, we destroy its native resources. To achieve that we need to make following changes:

1. Introduce reference from object to its j.l.Class instance. 2. Class registry - introduce references from j.l.Classes to its defining j.l.Classloader and references from j.l.Classloader to j.l.Classes loaded by it (unloading is to be done for j.l.Classloaderand corresponding j.l.Classes at once).

Introduce reference from object to its j.l.Class instance.

DRLVM has definite implementation specifics. Object is described with native VTable structure, which has pointers to class and other related data. VTables can have different sizes according to object class specifics. The main idea of referencing j.l.Class from object is to make VTable a special Java object with reference to appropriate j.l.Class instance, but give it a regular object view from GC point of view. VTable pointer is located in object by zero offset and therefore can be simply considered as reference field. Thus we can implement j.l.Class instance tracing from object via VTable object. VTable object is considered to be pinned for simplification.

In summary, having class registry and reference from object to its j.l.Classinstance we guarantee that some class loader CL can be unloaded only if three conditions are fulfilled described above (*). To find out when Java part of class loader was unloaded j.l.Classloader instance should be enumerated as weak root. When this root becomes equal to null – destroy native memory of appropriate class loader.

Pros:

- Unification of unloading approach – no additional requirements from GC.
- Stop-the-world is not required.
- GC handles VTables automatically as regular objects.

Cons:

- Number of objects to be increased. (by factor of 0.2% avg. for dacapo+jvm98+jbb)
- Memory footprint to be increased both for native and Java heaps (as VTable objects appear).
- Additional test+mark operation for each object in heap during gc.

"Generational" vtable marks design (Etienne)

1- During normal operation, the VM keeps hard references to all class loader instances. [This prevents any premature class loader death].

2- At the start of an epoch (or just before), all vtable bits (or byte or word) are cleared. [From now on, I will use the "bit" terminology for simplicity. The bit may reside in an otherwise unused byte or even word, for efficiency purpose].

3- The end of an epoch happens "no sooner" than when all generations /
heap parts have been collected at least once since the epoch start.
[One can cheat and visit objects of uncollected parts/generations to mark their vtables].

4- An "old generation" collection is chosen as the end of an epoch. This is "end of epoch collection". [As class loaders/classes are likely to have moved to older generations, there's no point trying to kill them in young collections].

5- Just before starting the "end of epoch collection", all the class-loader vtable lists are visited (and bits are cleared in prevision of the next epoch). All vm references to [candidate] class loaders with no surviving objects (nor active methods) (e.g. no vtable bit set) are made "weak".

6- The "end of epoch collection" is launched.

7- There's actually no need for "rescuing" class loaders. The vm reference to any surviving [candidate] class loader is made hard again.
Interesting fact: other candidate class loaders cannot have any instance (nor any active method) as GC doesn't create instances nor method calls. So, there's no need for a rescuing dance! The list of dying class loaders can be used for freeing related native resources.

IMO: simple, clean, efficient...

"Finalization-like" vtable marks design (Etienne, Salikh)

1. clean vtable marks before "class unloading" collection 2. enumerate classloader roots as weak and collect array of user classloader pointers for later use – let's call it "unload list" 3. trace the heap 4. scan vtable marks and "revive" marked class loaders, by adding the strong root from the previously collected "unload list". Remove the revived classloaders from unload list. 5. repeat steps (3) and (4) until there is no classloaders to revive. Second and subsequent traces start from newly added roots and only scan objects that were not reached during previous traces. 6. unload the classloaders, pointed by the "unload list" – this reclaims native resources 7. let the GC finish collection and reclaim unreachable objects – this reclaims java objects

This design unloads classloaders at the end of the very same epoch when they became unloadable.

To implement this design, a new GC-VM interaction (`vm_trace_complete()`) will be needed:

```
     GC                                                    VM
      |--------------------------------------> vm_enumerate_root_set()
      | gc_add_root_set_entry()<-------------------------------|
      | gc_add_root_set_entry()<-------------------------------|
      | gc_add_root_set_entry()<-------------------------------|
      |<- - - - - - - - - -return from vm_enumerate_root_set()
      |                                                       |
[trace heap]                                                  |
      |                                                       |
      |--------------------------------------> vm_trace_complete()
      | gc_add_root_set_entry()<-------------------------------|
      | gc_add_root_set_entry()<-------------------------------|
      |< - - - - - - - - - - return from vm_trace_complete()-|
      |                                                       |
[trace heap from new roots,                                   |
  if there are any]                                           |
      |--------------------------------------> vm_trace_complete()
      |< - - - - - - - - - - return from vm_trace_complete()-|
      |
[no retrace, as no new roots were received]
      |
[reclaim space]
      |
```

Additionally, even finalization itself can be moved out of GC responsibility, using this interface and one additional function to query if the object was already reached or not.

"Explicit trace" design ("Mark and scan based approach") (Alexey)

Java heap trace is performed by VM Core at the beginning of stop-the-world. VTables, classes and classloaders are traced transitively from each object. If some class loader and its classes are unreachable and there is no object of these classes, then exclude this class loader from enumeration to make GC collect it. After GC happens and appropriate j.l.Classloader instance is collected – remove native resources from C heap: class loader and all classes loaded by it, jitted code and so on. Corresponding Java objects should already be collected by GC at this moment.

Pros:

- Simplicity – requires only additional mark&scan functionality on VM side to detect classes for unloading + few changes in enumeration algorithm.

Cons:

- Requires additional GC/VM functionality to trace j.l.Class and j.l.Classloader instances from each object.
- Duplicates mark&scan functionality on VM side.
- Affects every plugged GC.
- "Stop-the-world" state of VM is required, i.e. all threads except the one performing unloading should be suspended.
- Possibly some additional limitations on new GCs.

Secondary root set design (Robin)

The efficiency problem with the 'Automatic Vtable' design is that a test-and-mark operation per vtable pointer is costly. The unconditional mark addresses this issue, but leaves the problem of finding reachable class/loader objects. This algorithm attempts to combine the efficiency of the first with the simplicity of the second.

This proposal requires

- A cheap mechanism for determining vtable reachability (eg unconditional byte mark)
- A mechanism for enumerating the vtables (such as a linked list or table)

The algorithm proceeds as follows:

- VM maintains a weak reference to every ClassLoader object
- At the end of the heap closure (and before processing reference types), enumerate the vtables. Add the j.l.Class object of any marked vtable to the root set.
- Trace from this root set using standard GC trace mechanism.
- Perform reference type processing. Unreachable classloaders will be freed, and non-java resources cleaned up using one of the standard mechanisms

The second trace operation should be cheap enough that it can be performed at the end of every major GC. Optionally this cost could be reduced by combining with one of the above mechanisms (eg VM usually maintains root ref to classloader, and demotes to weak if it wants to attempt class unloading).

Pros:

- Low per-GC overhead
- Relative simplicity

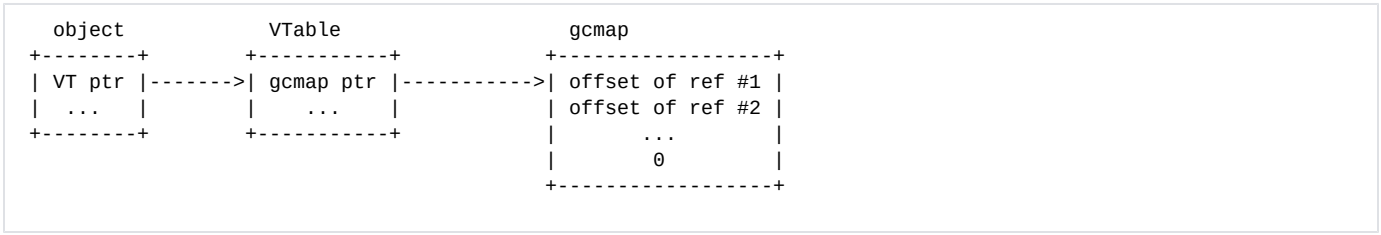
Cons:

- Requires secondary mark mechanism for vtables

Technical details on current DRLVM design

Object GC maps

In the current GC-VM interface, which is used in DRLVM (see vm/include/open/gc.h and vm/include/open/vm_gc.h), the GC never asks VM about gcmap; instead, it is building a gcmap itself as one of the class loading steps. VM calls `gc_class_prepared()`
for each loaded class, and GC uses various query functions to learn about types and offsets of object fields. The GC then stores the constructed gcmap pointer into the GC-private field of `VTable`.



(* actually, in the current default collector "gc_cc", gcmap ptr also has some flags in lower 3 bits, and gcmap has some fields before offsets array as well *)