CompressedOops

Compressed oops in the Hotspot JVM

What's an oop?

An "oop", or "ordinary object pointer" in HotSpot parlance is a managed pointer to an object. It is normally the same size as a native machine pointer. A managed pointer is carefully tracked by the Java application and GC subsystem, so that storage for unused objects can be reclaimed. This process can also involve relocating (copying) objects which are in use, so that storage can be compacted.

The term "oop" is traditional to certain VMs that derive from Smalltalk and Self, including the following:

- Self (an prototype-based relative of Smalltalk) https://github.com/russellallen/self/blob/master/vm/src/any/objects/oop.hh
- Strongtalk (a Smalltalk implementation) http://code.google.com/p/strongtalk/wiki/VMTypesForSmalltalkObjects
- Hotspot http://hg.openjdk.java.net/hsx/hotspot-main/hotspot/file/0/src/share/vm/oops/oop.hpp
- V8 http://code.google.com/p/v8/source/browse/trunk/src/objects.h (mentions "smi" but not "oop")

(In some of these systems, the term "smi" refers to a special non-oop word, a pseudo-pointer, which encodes a small, 30-bit integer. This term can also be found in the V8 implementation of Smalltalk.)

Why should they be compressed?

On an LP64 system, a machine word, and hence an oop, requires 64 bits, while on an ILP32 system, oops are only 32 bits. But on an ILP32 system there is a maximum heap size of somewhat less than 4Gb, which is not enough for many applications. On an LP64 system, though, the heap for any given run may have to be around 1.5 times as large as for the corresponding ILP32 system (assuming the run fits both modes). This is due to the expanded size of managed pointers. Memory is pretty cheap, but these days bandwidth and cache is in short supply, so significantly increasing the size of the heap just to get over the 4Gb limit is painful.

(Additionally, on x86 chips, the ILP32 mode provides half the usable registers that the LP64 mode does. SPARC is not affected this way; RISC chips start out with lots of registers and just widen them for LP64 mode.)

Compressed oops represent managed pointers (in many but not all places in the JVM) as 32-bit values which must be scaled by a factor of 8 and added to a 64-bit base address to find the object they refer to. This allows applications to address up to four billion *objects* (not bytes), or a heap size of up to about 32Gb. At the same time, data structure compactness is competitive with ILP32 mode.

We use the term *decode* to express the operation by which a 32-bit compressed oop is converted into a 64-bit native address into the managed heap. The inverse operation is *encoding*.

Which oops are compressed?

In an ILP32-mode JVM, or if the UseCompressedOops flag is turned off in LP64 mode, all oops are the native machine word size.

If UseCompressedOops is true, the following oops in the heap will be compressed:

- the klass field of every object
- every oop instance field
- every element of an oop array (objArray)

The Hotspot VM's data structures to manage Java classes are not compressed. These are generally found in the section of the Java heap known as the Permanent Generation (PermGen).

In the interpreter, oops are never compressed. These include JVM locals and stack elements, outgoing call arguments, and return values. The interpreter eagerly decodes oops loaded from the heap, and encodes them before storing them to the heap.

Likewise, method calling sequences, either interpreted or compiled, do not use compressed oops.

In compiled code, oops are compressed or not according to the outcome of various optimizations. Optimized code may succeed in moving a compressed oop from one location in the managed heap to another without ever decoding it. Likewise, if the chip (i.e., x86) supports addressing modes which can be used for the decode operation, compressed oops might not be decoded even if they are used to address object fields or array elements.

Therefore, the following structures in compiled code can refer to either compressed oops or native heap addresses:

- register or spill slot contents
- oop maps (GC maps)
- debugging information (linked to oop maps)
- oops embedded directly in machine code (on non-RISC chips like x86 which allow this)
- nmethod constant section entries (including those used by relocations affecting machine code)

In the C++ code of the HotSpot JVM, the distinction between compressed and native oops is reflected in the C++ static type system. In general, oops are often uncompressed. In particular C++ member functions operate as usual on receivers (*this*) represented by native machine words. A few functions in the JVM are overloaded to handle either compressed or native oops.

Important C++ values which are never compressed:

- C++ object pointers (*this*)
- handles to managed pointers (type Handle, etc.)
- JNI handles (type jobject)

The C++ code has a type called narrowOop to mark places where compressed oops are being manipulated (usually, loaded or stored).

Using addressing modes for decompression

Here is an example of an x86 instruction sequence that uses compressed oops:

```
! int R8; oop[] R9;  // R9 is 64 bits
! oop R10 = R9[R8];  // R10 is 32 bits
! load compressed ptr from wide base ptr:
movl R10, [R9 + R8<<3 + 16]
! klassOop R11 = R10._klass;  // R11 is 32 bits
! void* const R12 = GetHeapBase();
! load compressed klass ptr from compressed base ptr:
movl R11, [R12 + R10<<3 + 8]
```

Here is an example of a sparc instruction sequence which decodes a compressed oop (which might be null):

```
! java.lang.Thread::getThreadGroup@1 (line 1072)
! L1 = L7.group
ld  [ %l7 + 0x44 ], %l1
! L3 = decode(L1)
cmp  %l1, 0
sllx  %l1, 3, %l3
brnz,a   %l3, .+8
add  %l3, %g6, %l3  ! %g6 is constant heap base
```

(Annotated output is from the disassembly plugin.)

Null processing

A 32-bit zero value decodes into a 64-bit native null value. This requires an awkward special path in the decoding logic, to the point where it is profitable to statically note which compressed oops (like klass fields) are guaranteed never to be null, and use a simpler version of the full decode or encode operation.

Implicit null checks are crucial to JVM performance, in both interpreted and compiled bytecodes. A memory reference which uses a short-enough offset on a base pointer is sure to provoke a trap or signal of some sort if the base pointer is null, because the first page or so of virtual address space is not mapped.

We can sometimes use a similar trick with compressed oops, by unmapping the first page or so of the virtual addresses used by the managed heap. The idea is that, if a compressed null is ever decoded (by shifting and adding to the heap base), it can be used for a load or store operation, and the code still enjoys an implicit null check.

Object header layout

An object header consists of a native-sized mark word, a klass word, a 32-bit length word (if the object is an array), a 32-bit gap (if required by alignment rules), and then zero or more instance fields, array elements, or metadata fields. (Interesting Trivia: Klass metaobjects contain a C++ vtable immediately after the klass word.)

The gap field, if it exists, is often available to store instance fields.

If UseCompressedOops is false (and always on ILP32 systems), the mark and klass are both native machine words. For arrays, the gap is always present on LP64 systems, and only on arrays with 64-bit elements on ILP32 systems.

If UseCompressedOops is true, the klass is 32 bits. Non-arrays have a gap field immediately after the klass, while arrays store the length field immediately after the klass.

Zero based compressed oops

Compressed oops use an arbitrary address for the narrow oop base which is calculated as java heap base minus one (protected) page size for implicit NULL checks to work. This means a generic field reference is next:

```
<narrow-oop-base> + (<narrow-oop> << 3) + <field-offset>.
```

If the narrow oop base can be made to be zero (the java heap doesn't actually have to start at offset zero), then a generic field reference can be just next:

```
(<narrow-oop << 3) + <field-offset>
```

Theoretically it allows to save the heap base add (current Register Allocator does not allow to save register). Also with zero base the null check of compressed oop is not needed.
Current code for decoding compressed oops looks like this:

```
if (<narrow-oop> == NULL)
    <wide_oop> = NULL
else
    <wide_oop> = <narrow-oop-base> + (<narrow-oop> << 3)
```

With zero narrow oop base the code is much simpler. It needs only shift to decode/encode a compressed oop:

```
<wide_oop> = <narrow-oop> << 3
```

Also if java heap size < 4Gb and it can be moved into low virtual address space (below 4Gb) then compressed oops can be used without encoding /decoding.

Zero based implementation tries to allocated java heap using different strategies based on the heap size and a platform it runs on.
First, it tries to allocate java heap below 4Gb to use compressed oops without decoding if heap size < 4Gb.
If it fails or heap size > 4Gb it will try to allocate the heap below 32Gb to use zero based compressed oops.
If this also fails it will switch to regular compressed oops with narrow oop base.