VirtualCalls

When an invokevirtual call is linked, the linker resolves the call to an abstract target method. This boils down to a target class and a vtable index within that class. Since the target class is statically guaranteed by the verifier, the significant result (for execution of the call) is the vtable index.

A vtable is a display, associated with a class, of actual target methods for every virtual method which the class responds to, or any of its superclasses responds to. Just as the fields of an individual instance grow from a prefix shared with super classes toward a suffix unique to the instance's exact class, the vtable of a class grows from a prefix of method slots shared with super classes toward a suffix of methods which are unique to the class. If the class has no subclasses, those methods, though virtual, might not be overridden. But subclasses may replace those inherited prefix slots with their own overriding methods.

If a class is concrete, all of its vtable entries refer to concrete (executable) methods. If a class is abstract, some of its vtable entries may lack executable code. (They are occupied by abstract methods. If such an entry gets executed by accident, an error is thrown.)

In HotSpot, a class's vtable is allocated directly inside the InstanceKlass object which serves as metadata for the class. This means that a virtual call can proceed in two indirections: One to get the _klass field from the header of an object, and another to load the vtable slot from the InstanceKlass. (A typical offset is 200 or 300 bytes into the middle of the InstanceKlass.) After the vtable, the InstanceKlass contains other variable-sized structures, such as the itable (which is similar to the vtable) and oops maps for instance fields. This double indirection pattern is very similar to that seen in most C++ implementations. However, there is (currently) a third indirection to fetch an entry point from the Method, pointer to which occupies a vtable slot. For example, see the code in src/cpu/x86/vm/vtableStubs_x86_32.cpp.

It is legal for an invokevirtual bytecode to refer to a final method. A final method need not have a vtable slot allocated. This means that, after linking, an invokevirtual bytecode might in fact collapse into the equivalent of an invokestatic bytecode. The interpreter is prepared to do this.

The Java language exposes two states for call sites: Unlinked and linked. Once a virtual call site is linked it is handled by a vtable-based calling sequence (or a direct one, in rare cases).

Call sites in compiled code typically do *not* use the vtable-based calling sequence. That is because most of them are "well behaved" and can be implemented more efficiently than via a triple indirection. Even if the compiler does not take special steps to optimize a call site, there is an intermediate state, between unlinked and vtable-based, where a call site can point to a single method. This state is called "monomorphic". A monomorphic call site is one which optimistically points to the only concrete method that has ever been used at that particular call site. The calling sequence consists of verifying that the current (dynamically determined) receiver is exactly the same type as the previous call. That check is a simple pointer comparison, of a constant versus the InstanceKlass* loaded (via one indirection) from the object header. If the check succeeds, the call is made directly.

The receiver class is the one observed when the call site was first linked. If a new class is ever encountered (and this eventually happens for roughly ten percent of call sites), the call site is relinked to use the vtable based calling sequence: The address of the call instruction is patched to a vtable dispatch stub. This relinking is invisible to the Java programmer.

More details: Vtable stubs are assembled on the fly, but there are only about a hundred of them. There is one for each distinct vtable slot (offset within the InstanceKlass of the second indirection). For a monomorphic call site, the caller loads the expected class (as a InstanceKlass*) into a standard (non-argument) register, and optimistically jumps to the expected target method. All such target methods are compiled with a special "verified entry point" (VEP) which includes an instruction to compare the incoming InstanceKlass* against the receiver's _klass field. If the check succeeds, control falls through to the normal "unverified entry point" (UEP) of the compiled method. If the check fails, control branches out of the (wrong) method into a stub which performs an up-call to the JVM, to figure out what's wrong, and (usually) relink the call site.

Sample Code

Since most call sites are monomorphic, they can be completed in a single branch with a parallel receiver type check. Here is a generic instruction trace of this simple case:

monomorphic call to a virtual or interface method

```
callSite:
    set #expectedKlass, CHECK
    call #expectedMethod.VEP
---
expectedMethod.VEP:
    cmp (RCVR + #klass), CHECK
    jump,ne wrongMethod
compiledEntry:
    ...
```

Any method that can be a target of such a call has two entry points, the VEP ("verified entry point") and the UEP ("unverified entry point"). The former does a two-instruction type check and then falls into the latter.

Here is a generic instruction trace of a polymorphic interface call. It is the equivalent of the C++ virtual function dispatch.

polymorphic call to a virtual method

```
callSite:
    set #garbage, CHECK
    call #vtableStub[vtableSlot]
---
vtableStub[vtableSlot]:
    load (RCVR + #klass), TEM
    load (TEM + #(vtable + vtableSlot)), METHOD
    load (METHOD + #compiledEntry), TEM
    jump TEM
---
compiledEntry:
    ...
```

In all, that is 3 memory references and two nonlocal jumps.

Note that the intermediate "vtable stub" is customized to the vtable offset (which is usually in the 200-300 range, given that the header of a InstanceKlass is around 200 bytes, or somewhat more on LP64). The vtable stub is not customized to the klass.

The final memory reference in this instruction trade could be removed by storing a pointer to each method's UEP in vtable (and itable) entries, as C++ does. This has not been done because it would then be difficult for the interpreter to use the vtables. (An earlier version of the system had two-word vtables, to display both kinds two entry points on an equal footing, but this led to complexities and, worst of all, race conditions.) It is an interesting problem to try to invert the preference the current design gives to the interpreter; the interpreted entry point would have to be hidden somewhere at a fixed offset from the method's compiled entry point, but not close enough to disturb the tuning of the VEP.

Call sites of this polymorphic form are rarely generated directly by the JIT; instead, they are generated to call to a bootstrapping linkage routine, which then sets them (if all goes well) to the monomorphic state described above. When the second receiver type is encountered, the linkage code (at the `wrongMeth` od jump target above) is called to transition the call site to its polymorphic state. The state changes are summarized in the leading comments of src/share /vm/code/compiledIC.hpp.