InterfaceCalls

When an invokeinterface call is linked, the linker resolves the call to an abstract target method, in an interface. This boils down to a target interface and a so-called *itable index* within that interface.

Target interfaces are never statically guaranteed by the JVM verifier; every invokeinterface receiver is typed as a simple object reference. Therefore (unlike invokevirtual calls), no assumptions can be made about the receiver's vtable layout. Instead, the receiver's class (as represented by its _klass field) must be checked more carefully. Where a virtual call can blindly perform two or three indirections to reach the target method, an interface call must first inspect the receiver's class to determine (a) if that class actually implements the interface, and (b) if so, where that interface's methods are recorded within that particular class.

There is no simple prefixing scheme in which an interface's methods are displayed at fixed offsets within every class that implements that interface. Instead, in the general (non-monomorphic) case, an assembly-coded stub routine must fetch a list of implemented interfaces from the receiver's InstanceKlass, and walk that list seeking the current target interface.

Once that interface is found (within the receiver's InstanceKlass), things get a little easier, because the interface's methods are arranged in an *itable, or "interface method table"*, a display of methods whose slot structure is the same for every class that implements the interface in question. Therefore, once the interface is found within the receiver's InstanceKlass, an associated offset directs the assembly stub to an itable embedded in the InstanceKlass (just after the vtable, as one might expect). At that point, invocation proceeds as with virtual method calls.

In the interpreter, the state of a linked invokeinterface instruction consists of two words (both in the constant pool cache): The interface being sought, and the index within that interface's itable of the method being called. (There is no need to search for symbolic method names as in message-oriented languages; only the interface needs to be searched for.) In some very rare corner cases (as with invokevirtual), the linkage state of an invokeinterface instruction might actually direct the interpreter to treat the call equivalently to an invokespecial or invokevirtual. That latter case can occur when an interface declares an Object method like hashCode as an interface method; the JVM does not allocate itable slots for those methods.

Nearly the same optimizations apply to interface calls as to virtual calls. As with virtual calls, most interface calls are monomorphic, and can therefore be rendered as direct calls with a cheap check.

It is a curious fact that the searching process described above, while currently a linear search over an array embedded in the receiver's InstanceKlass, could also be implemented, with roughly equivalent performance, as a pointer-chasing search over a linked-list representation of implemented interfaces. Indeed, some languages use pointer chasing for dynamic lookup. The advantage of such a representation is a looser coupling between a class and its methods (or in JVM terms, its itables; an itable may contain either single methods or groups of methods). The looser coupling would allow a class to extend its implemented interfaces in a type-safe and compatible manner, just as a SmallTalk class can add new methods with a modest amount of pointer swapping.

Sample Code

Since most call sites are monomorphic, they complete very quickly; see the monomorphic sample code under VirtualCalls.

Here is a generic instruction trace of a polymorphic interface call.

polymorphic call to an interface method

```
callSite:
    set #calledInterface, CHECK
    call #itableStub[itableSlot]
---
itableStub[itableSlot]:
    load (RCVR + #klass), KLASS_TEM
    load (KLASS_TEM + #vtableSize), TEM
    add  (KLASS_TEM + TEM), SCAN_TEM
tryAgain:
        # this part is repeated zero or more times, usually zero
        load (SCAN_TEM + #itableEntry.interface), TEM
        cmp TEM, CHECK
        jump,eq foundInterface
        test TEM
        jump,z noSuchInterface
        inc #sizeof(itableEntry), SCAN_TEM
        jump tryAgain
tryAgain:
    load (SCAN_TEM + #itableEntry.interface), TEM
    cmp TEM, CHECK
    jump,eq foundInterface
foundInterface:
    load (SCAN_TEM + #itableEntry.offset), TEM
    load (KLASS_TEM + TEM + #itableSlot), METHOD
    load (METHOD + #compiledEntry), TEM
    jump TEM
---
compiledEntry:
    ...
```

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

Note that the intermediate "itable stub" is customized to the itable offset (which is always small and often zero). It is not customized to the interface; the call site uses the CHECK register (also used by monomorphic calls) to transmit the desired interface.

For some other approaches to reducing the cost of this operation, see Faster Interface Invocation in the Da Vinci Machine Project pages.

(The alert reader will note that updating the call site from its monomorphic state to its polymorphic state requires a two-word MP-safe transaction. This is simulated by "bungee patching", in which the target of the call is temporarily patched to a safely initialized copy of the desired call site. This extra indirection is later retracted at a safe moment, by the routine `ICStub::finalize`.)