FastInterfaces

Patch name: fasti.patch

The current interface calling sequences are documented in HotSpotInternals.

Most implementors of interfaces implement a single interface, and over half of those interfaces have three or fewer methods. (See bug 6580709 for more information.) We will allocate a tunable number (`-XX:FastInterfaceMethods`, default four) to control special treatment of these cases.

Sketch of Implementation

Every method will be given a new oop slot called `implements`. This slot will contain either a null, meaning that the method does not implement any interface, a pointer to a klassOop for an interface, meaning that it implements that interface and no others, or a pointer to `object_klass`, meaning that the method implements two or more interfaces. If the same method (name and signature) shows up in two interfaces related by subtyping, only the occurrence in the supertype will ever be counted.

(For example, `List.isEmpty` will never be given an itable slot, since that method is defined earlier in `Collection.isEmpty`. Also, Object methods never show up in itables at all.)

When an `instanceKlass` is being laid out, four (or `FastInterfaceMethods`) vtable slots will be allocated after the 3 vtable slots for `Object` (`equals`, `hashCode`, `finalize`). The default entries for those slots will a sentinel value (probably null) that denotes a normal itable stub.

When the itables are being filled in, if itable slot N (N < 4) is defined by a unique method, then a entry point for that method is inserted into the corresponding reserved vtable entry (3+N). If no methods implement an itable slot N (for any interface) then the default entry in vtable entry (3+N) is left undisturbed.

If slot N of two or more interfaces is implemented, then an arbitrary choice is made to speed up one or the other of the calls. It would be wise to consult profile data to decide which to favor.

The specialized entry point for a method in slot N will perform these actions:

1. assume a special register CHECK (RCX on x86) contains the target interface
2. assume a special register RECV (RAX on x86) contains the receiver argument
3. load the receiver class
4. load vtable slot (3+N) of the class into the METHOD register
5. if it is the sentinel value, branch to the slot itable stub
6. compare the `implements` field of the method against the target interface
7. if they are unequal, branch to the slow itable stub
8. otherwise, jump into the method's verified entry point

(Note that a carefully constructed sentinel value can fold the two checks into one check. E.g., the sentinel value could be a dummy method with a never-matching `implements` field. Also, the special entry point should be made part of the nmethod, if the method gets compiled. It could be placed just before the unverified entry point, to get good prefetching from the CPU. If this is done, then the compiled entry point of the dummy method will have to contain a copy of the stub code above.)

The assignment of vtable slots (3+N) to itable methods is flexible. The numbering amounts to a globally defined hash code from (interface, itable slot) to the range (0..`FastInterfaceMethods`). The hash may need tuning so as to allow the hot methods of `Collection` to co-exist with those of `List` and `Ma`
.

When an interface call site is promoted to the megamorphic state (i.e., it is properly polymorphic), the linkage routine should determine which vtable slot applies to the desired combination of interface and itable slot. If there is none, the normal itable stub is used. If there is one, it is unique, and a vtable-style stub is used instead, one which implements the specialized entry sequence described above.

Implementation Status

Idle. (No sponsor at present.)