Overview of CompiledIC and CompiledStaticCall

There are two primary flavors of compiled call sites in the JVM, CompiledStaticCall and CompiledIC. CompiledStaticCall is pretty much what it sounds like, a compiled call site where the target method is a static Java method. It can only be in one of three states: clean, to interpreted and to compiled. In the clean state it points at SharedRuntime::get_resolve_static_stub(). In the compiled state state it points to the verified entry point of the nmethod. The interpreted state is more complicated because the interpreter needs to have extra state passed in to perform the call. The call site points to a trampoline called the static call stub that the compiler generated when it emitted the original call. The trampoline looks like this:

mov method, method_reg
jmp entry

method is the methodOop of the static method being invoked and method_reg is a platform dependent register. On sparc it's G5 and on x86 it's rbx. Entry is the c2i entry point of the method being called.

A CompiledIC is used for instance method invocations. It has two variants, the optimized virtual and the virtual call. The virtual call is used when a call site appears to have multiple possible target methods and the optimized virtual in used in cases where there is a single target method. The optimized virtual case is very much like a CompiledStaticCall.

The virtual case is fairly complicated. The call site consists of an instruction that loads a constant into a register and a call. The target of the call can be various different functions depending on the state of the call and the constant value is used by these functions to validate the invocation. The possible state of a CompiledIC are clean, monomorphic and megamorphic. There are additional states called transition states that have to do with patching running code in a MT safe way, but that doesn't add any behaviours.

A CompiledIC call site looks like this:

mov value, cache_reg
call entry

cached_reg is G5 on sparc and rax on x86. A clean call site starts out with value as Universe::non_oop_word and entry pointing at either SharedRuntime::get_resolve_opt_virtual_call_stub or SharedRuntime::get_resolve_virtual_call_stub depending on which kind it is. The non_oop_word value is a special number that helps with the MT safe patching of these call sites. It's commonly -1.

A call site may appear to have multiple possible receiver methods during compilation but in practice it's common that only a single one will actually be called. This is the monomorphic state and the primary purpose of a CompiledIC is to make that case fast since it's the most common.

The first time a clean call site is invoked the JVM looks at the type of the receiver and looks up the method that would be invoked. Because the call site is clean the CompiledIC assumes that the type of the current receiver is a good guess for the future type, so it points the call at the unverified entry point of the method. This is a special entry point with an inline cache check that verifies that the current receiver matches the receiver recorded in the call site. If it does then it continues into the verfified entry point and executes the method. If the method being invoked has compiled code then the inline cache check is part of the nmethod and looks roughly like this:

cmp reg, [receiver_reg + klass_offset]
jne SharedRuntime::get_ic_miss_stub()

The value in receiver_reg is the klassOop of the original receiver and the compare is a quick type equality check to ensure that the method is compatible with the current type.

In the case where the method doesn't have compiled code then the entry point is the unverfied entry point of the method adapter for this method. Again because the interpreter calling convention is different from compiled the value in the CompiledIC is a more complicated value, consisting of a pair of the methodOop to be invoked and the klassOop of the original receiver. Otherwise it operates very similarly to the compiled case.

If the types don't match then this is considered an inline cache miss and the code calls into the runtime to resolve against the new type. This commonly results in the call site going megamorphic and using a C++ vtable style dispatch to get to the right method.

The extra complexity in this code comes for performing MT-safe patching of call sites where you might need to change two values. This means that certain kinds of call site transitions can safely be done in place while others have to use the ICBuffer machinery. MT-safe patching of call sites requires that it's possible to change the destination of a call while another thread is possibly executing the call site. On architectures where instructions are fixed size this is easy to accomplish since writing a single instruction is usually sufficient to change the destionation of a call. On x86 it's more complicated because of the variable length encoding of instructions. To handle this Hotspot forces the destination portion of the encoding to be aligned on a 4 byte boundary. This makes it possible to update the call site using a single memory operation, ensuring that a mix of the old and new address is never seen.

The value portion of the CompiledIC can't be written atomically relative to the call destination so transitions are only considered safe it the new destination is safe with respected to seeing a sheared value. For instance when a call site is clean, the value portion is set to -1. When transitioning to the monomorphic call site to value becomes a klassOop. Since the unverified entry point is only using the value in a pointer compare, it won't generate false positives if it happens to see a sheared value. It will think it had a miss but then it will call into IC miss code and check the call site under a lock where the call site will appear to be in the correct state, so it won't treat it as a miss.

Other call paths will actually load from the value being passed through and in those cases seeing a sheared value would cause a crash. To patch those safely, a copy of the CompiledIC is generated out of line in an ICBuffer and the entry point is patched to point at this new call site trampoline. Since the ICBuffer is new instruction space it's impossible for a thread to anything but the new value. During the next safepoint the contents of the ICBuffer are copied back into the original call site, safely setting it up with the right value and entry point.