# Runtime Infrastructure - Feature #6939

## improve proxy performance by hard-coding method invocation

11/16/2022 02:48 PM - Eric Faulhaber

| | | | |
|---|---|---|---|
| **Status:** | New | **Start date:** | |
| **Priority:** | Normal | **Due date:** | |
| **Assignee:** | | **% Done:** | 0% |
| **Category:** | | **Estimated time:** | 0.00 hour |
| **Target version:** | | | |
| **billable:** | No | **vendor_id:** | GCD |

**Description**

**Related issues:**

| | |
|---|---|
| Related to Base Language - Feature #6819: refactor FWD proxy implementation t... | **New** |

### History

#### #1 - 11/16/2022 02:48 PM - Eric Faulhaber

*- Related to Feature #6819: refactor FWD proxy implementation to use ReflectASM instead of Java Method reflection added*

#### #2 - 11/16/2022 03:28 PM - Eric Faulhaber

When I originally implemented the com.goldencode.proxy package, the pressing goal at the time was to improve memory consumption compared to the default Java proxy implementation. Specifically, java.lang.reflect.Proxy did not allow a superclass to be specified, and several of our use cases could benefit from many of the methods specified in proxied interfaces being implemented in a base class, allowing only those not implemented in the base class to be proxied. This allowed for a much lower PermGen footprint, since all those base class methods did not need to have duplicative proxy methods defined in the proxy class.

Anyway...the rest of the design was very similar to that of java.lang.reflect.Proxy, in that it used an InvocationHandler, to which a Method object, an Object instance on which to invoke that method, and an optional argument array are passed at runtime. Most implementations of that invocation handler initially used reflection to manage the actual method invocation. Later, we switched most of those to use ReflectASM instead, for better performance. But, recently we discovered that ReflectASM actually can perform worse than reflection on certain hardware (Constantin, could you provide a link to a summary of those findings?).

So, a fundamental problem with the proxy implementation is the performance of the ultimate invocation of the "actual" method on the object which backs the proxy. The point of a proxy in most of our use cases is to employ an interceptor pattern to inject processing between the invocation of a method call on the proxy and its eventual invocation on a method of the same signature on the backing object. Today, the interceptor is the code which implements the InvocationHandler interface, to perform the behavior we want to inject. The current design delegates the ultimate invocation of the method on the backing object to this interceptor as well.

It seems we can eliminate the relatively poor performance of both reflection and ReflectASM (and the need to choose between them, based on hardware) by simply relieving the interceptor implementation of the obligation to invoke the method on the backing object. Instead of delegating this responsibility to an InvocationHandler implementation, the method invocation on the backing object can be achieved with a few bytecode instructions in the proxy methods themselves. Since we know the signature of the method being invoked (it is the same as the signature of the method proxy), the invocation on the backing object could be hard-coded in the proxy method, which would be much faster than either reflection or ReflectASM.

There still will be a callback to the interception logic; however, that logic would not perform the final method invocation. The API used to invoke the interception callback would need to provide a means for that callback to inform the proxy method whether or not the ultimate invocation on the backing object should be made, possibly through a return value. I haven't figured out the details of this API yet, and in fact it might need to be flexible, based on the use case.

Since some existing InvocationHandler implementations (e.g., RecordBuffer$Handler.invoke) have several points at which they may invoke the method on the backing object, we will need to deal with this requirement in some way. We have the flexibility of the ProxyAssemblerPlugin to enable custom method proxy implementations, but I haven't figured out the exact details of how we need to handle this refactoring yet.

#### #3 - 11/17/2022 07:41 AM - Constantin Asofiei

Eric Faulhaber wrote:

> But, recently we discovered that ReflectASM actually can perform worse than reflection on certain hardware (Constantin, could you provide a link to a summary of those findings?).

Is this what you are looking? https://proj.goldencode.com/issues/6819#note-3

#### #4 - 12/08/2022 01:07 PM - Constantin Asofiei

Eric, the main issue of the interceptor flavor of the InvocationHandler is that in some cases the target instance on which the method is invoked can be changed. So you can't just invoke the target method directly from the bytecode of the i.e. DMO Implementation class setter/getter. As the actual instance on which the method is invoked can be several layers of InvocationHandler deep.

#### #5 - 12/08/2022 01:10 PM - Constantin Asofiei

Constantin Asofiei wrote:

> ... directly from the bytecode of the i.e. DMO Implementation class setter/getter.

what I meant here is the Buffer proxy instance created for a DMO interface.

#### #6 - 12/08/2022 01:29 PM - Greg Shah

If we need the ability to temporarily replace the target methods for an implementation, we should consider a kind of dispatch table approach. Back in C, it was a common technique. Each function was represented by a function pointer and you would have a "table" of these function pointers which was really just an array of the pointers. You could swap out the array at any time because you actually stored it as a pointer to an array of pointers. You could backup the pointer to the pointer, replace it with a pointer to your version of the table (with different functions) and later on you could restore it back as needed. The replaced functions could even be smart enough to use the original table for some base functionality (e.g. like we might call super in an overridden method).

The core technique in fact was the basis for the C++ language implementation of virtual functions.

The point here is that we could do something similar with lambdas. The generated bytecode could store an array of lambdas and an modifiable reference to that array which we could potentially replace and restore as needed, controlling the specific set of methods being called.

#### #7 - 12/08/2022 01:43 PM - Constantin Asofiei

Greg Shah wrote:

If we need the ability to temporarily replace the target methods for an implementation,

I don't really understand what this would solve; the problem is not the target Method being invoked, but the target instance on which the Method is invoked.

#### #8 - 12/08/2022 01:55 PM - Greg Shah

In the lambda approach, the target instance is already captured, so changing the array also changes the target.

#### #9 - 12/12/2022 10:32 AM - Constantin Asofiei

*- File 6939_20221212a.patch added*

Eric, please review attached patch - it adds a invocation handler like this:

```
      public Object invoke(Object proxy,
                           final Method method,
                           BiFunction<Object, Object[], Object> f,
                           BiConsumer<Object, Object[]> c,
                           Object[] args)
```

Standalone tests show that it works as expected, but real apps are showing possible issues.

#### #10 - 12/12/2022 01:29 PM - Constantin Asofiei

Some additional notes:

- having both consumer and function lambdas doesn't really have a functional benefit, only function can be used and just return null in case of void methods.
- I need to test, but I think the number of emitted lambda$# synthetic Java methods can be reduced to just one, as all have the same signature, Object ref, Object[] args.

#### #11 - 12/21/2022 09:36 AM - Greg Shah

Do you have an updated version of the patch for 3821c?  It does not apply cleanly on 3821c rev 14461.

Eric: This still needs your review.

#### #12 - 12/21/2022 09:39 AM - Constantin Asofiei

Greg Shah wrote:

> Do you have an updated version of the patch for 3821c?  It does not apply cleanly on 3821c rev 14461.

I forgot to mention, the batch is built on top of 6129b/14341.  Do you want to port it to 3821c?

> Eric: This still needs your review.

#### #13 - 12/21/2022 09:42 AM - Greg Shah

Yes, I was planning for all performance work to go in there.

#### #14 - 12/21/2022 11:55 AM - Constantin Asofiei

*- File 6939_6819_lambdas_3821c_14461.patch added*

Greg Shah wrote:

> Yes, I was planning for all performance work to go in there.

OK, but not all performance work from 6129b ended up in 3821c, so we need to be careful if we use 3821c for profiling.

Attached patch is built on top of 3821c/14461, includes both #6819 and #6939 tasks.

#### #15 - 12/21/2022 01:19 PM - Constantin Asofiei

*- File deleted (6939_6819_lambdas_3821c_14461.patch)*

#### #16 - 12/21/2022 01:19 PM - Constantin Asofiei

*- File 6939_6819_lambdas_3821c_14461.patch added*

The correct patch is this, forgot to do bzr add before doing bzr diff.

### Files

| | | | |
|---|---|---|---|
| 6939_20221212a.patch | 25.1 KB | 12/12/2022 | Constantin Asofiei |
| 6939_6819_lambdas_3821c_14461.patch | 91.3 KB | 12/21/2022 | Constantin Asofiei |