Base Language - Feature #6819

refactor FWD proxy implementation to use ReflectASM instead of Java Method reflection

10/06/2022 02:43 AM - Constantin Asofiei

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Greg Shah	**% Done:**	50%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**vendor_id:**	GCD

Description

Related issues:

Related to Database - Feature #6895: improve performance of BufferReference m...	**New**
Related to Database - Support #4928: rewrite FieldReference to use lambdas in...	**New**
Related to Runtime Infrastructure - Feature #6939: improve proxy performance ...	**New**
Related to Base Language - Feature #7348: mutable proxy implementation	**New**

History

#1 - 10/06/2022 02:51 AM - Constantin Asofiei

- Subject changed from refactor FWD proxy implementation to use ReflectASM instead of Java Method reflection and other to refactor FWD proxy implementation to use ReflectASM instead of Java Method reflection

In a large application tested with 6129a, Utils.invoke(Method method, Object instance, Object... args) is being called some 8 million times, just for 2 very complex tests. These calls include both invocations like RUN statements and buffer proxy API calls, field getter/setter, etc.

I've ran the same tests without ReflectASM (uncommented the return method.invoke(instance, args); line in Utils.invoke and commented everything else), and performance was worse.

With ReflectASM usage in Utils.invoke, the penalty I see in the profiler is of MethodData.methodAccessIdx.get(idx) - this map lookup is done 8 million times.

We can refactor FWD proxy factory to emit a custom invocation handler instead of Java's, which accepts (beside the Java Method instance, as this is required to check at least the signature in RecordBuffer) a MethodAccess and index for ReflectASM - this will allow bypassing Java reflection.

Special attention is needed for non-public methods - IIRC, at least for private methods Java reflection is required.

#2 - 10/28/2022 04:36 AM - Eric Faulhaber

- Related to Feature #6895: improve performance of BufferReference methods buffer() and definition() added

#3 - 10/28/2022 01:50 PM - Constantin Asofiei

- File mthdtest.zip added

There is a tuning decision to be made if ReflectASM is better than Method.invoke:

- on a i7-1170K@3.6GHz, ReflectASM is 2xtimes better than Method.invoke
- on devsrv01, ReflectASM is 8xtimes worse
- on a i7-4790@3.6GHz, ReflectASM is 'only' 4xtimes worse

The test used for this is attached. Adjust the classpath in run.sh to the p2j.jar and execute it.

#4 - 11/16/2022 05:19 AM - Greg Shah

- Related to Support #4928: rewrite FieldReference to use lambdas instead of reflection added

#5 - 11/16/2022 02:48 PM - Eric Faulhaber

- Related to Feature #6939: improve proxy performance by hard-coding method invocation added

#6 - 11/16/2022 03:30 PM - Greg Shah

- Assignee set to Greg Shah

I'm working on some sample code to show a better approach.

#7 - 12/07/2022 09:04 PM - Greg Shah

- File mthdtest_converted_lambda_approach_20221207.zip added

I've been working on a different approach to this problem. The original problem noted above is that ReflectASM and Java reflection perform differently on different systems. It got me thinking about how slow those approaches are compared to direct usage of a lambda. In Java 8, there is a new facility called LambdaMetafactory which can be used to create a lambda given a Method instance. That lambda nearly equivalent to having a lambda hard coded in the source (i.e. "direct usage"). It struck me that using a lambda in this way would likely be significantly faster than either ReflectASM or Java reflection AND we already have access to the Method instance needed for the conversion.

I have made updates to the test program to show the new approach.

Here are the generated results:

```
      Scenario                ISG         SSG         IEG         SEG         ISS         SSS         IES         SES
---------------------- ----------- ----------- ----------- ----------- ----------- ----------- ----------- ----------
-
ReflectASM                  27.470      22.576      30.751      30.314      20.329      21.795      28.949      26.44
7
Utils.invoke               134.889     147.280     181.962     194.663     168.460     195.644     161.702     178.76
6
Java Reflection Method      25.831      89.949     113.715      73.243      78.653      53.857     114.970      81.10
5
lambda                       6.398       6.945      11.498       7.395       7.097       5.729       9.221       9.07
0
lambda from Method           6.981       5.307       8.669       8.385       5.637       5.297      10.776       8.32
9
direct method call           5.447       3.968       4.736       5.119       4.394       3.214       6.006       5.37
6
```

lambda is the use of the direct method reference (instance::method).

direct method call is the hard coded call to the method.

lambda from Method is the approach I'm suggesting where we use LambdaMetafactory to create the lambda.

The good news is that the result is in fact significantly faster than ReflectASM and Java reflection. The core worker is here:

```
   /**
    * Create a lambda of type T which represents the given implementation method and which is compatible
    * with the given functional interface.  The implementation method can be a static or instance method.
    * <p>
    * The implementation method that is passed is expected to only access local data or member data.  The
    * key implication of this is that a maximum of one variables will be captured.  If this is an instance
```

```
     * method, then the given instance parameter will be captured (bound) to the lambda.  Otherwise (for
     * static methods) there are no variables captured.
     * <p>
     * Given this code:
     * <p>
     * <pre>
     * public class Whatever
     * {
     *    public void setter(String s1, int subscript)
     *    {
     *    }
     * }
     * </pre>
     * <p>
     * and an instance of {@code Whatever} called {@code ref}, one can obtain a lambda referencing this setter
     * method which will be the the equivalent of a hard coded literal lambda {@code ref::setter}.  This code
     * would work:
     * <p>
     * <pre>
     *    Method impl  = Whatever.class.getDeclaredMethod("setter", new Class<?>[] { String.class, int.class })
;
     *    Method iface = BiConsumer.class.getMethod("accept", new Class<?>[] { Object.class, Object.class })
     *    BiConsumer<String, Integer> lamdba = Whatever.<BiConsumer<String, Integer>>lambdaFromMethod(impl, ifa
ce, ref);
     * </pre>
     * <p>
     * At that point, the resulting lambda could be called as {@code lambda.accept("param", 0)}.  Notice that
     * in this case the instance {@code ref} is already bound.  If {@code Whatever.setter()} is static, the
     * only difference in this is that {@code null} would be passed for the {@code instance} parameter.
     *
     * @param    impl
     *           The implementation method which will be called when the lambda is invoked.
     * @param    iface
     *           The method of the functional interface which is being implemented.
     * @param    instance
     *           The instance of the class which implements the method or {@code null} if the method is
     *           static.
     *
     * @return   The lambda for the given implementation method, typed as the provided functional interface
     *           method and already bound to the instance if provided.
     */
   public static <T> T lambdaFromMethod(Method impl, Method iface, Object instance)
   throws IllegalAccessException,
          LambdaConversionException,
          Throwable
   {
      MethodHandles.Lookup lookup = MethodHandles.lookup();

      String     ifaceName      = iface.getName();
      Class<?>   implCls        = impl.getDeclaringClass();
      Class<?>   ifaceCls       = iface.getDeclaringClass();
      Class<?>[] implParms      = impl.getParameterTypes();
      Class<?>[] ifaceParms     = iface.getParameterTypes();
      Class<?>   implRet        = impl.getReturnType();
      Class<?>   ifaceRet       = iface.getReturnType();
      MethodType invokedType    = (instance == null) ? MethodType.methodType(ifaceCls)             // static m
ethods take no captured parameter
                                                    : MethodType.methodType(ifaceCls, implCls);   // instance
 methods have a captured implicit first parameter which is the type of the instance
      MethodType samMethodType  = MethodType.methodType(ifaceRet, ifaceParms);
      MethodType instMethodType = MethodType.methodType(implRet, implParms);

      // Useful debugging output for understanding the inputs that are passed to the LambdaMetaFactory:
      // String     implName       = impl.getName();
      // String     p1             = Arrays.stream(implParms).map(t -> t.toString()).collect(Collectors.joinin
g(","));
      // String     p2             = Arrays.stream(ifaceParms).map(t -> t.toString()).collect(Collectors.joini
ng(","));
      // System.out.printf("Convert From %s %s.%s(%s)\n", implRet, implCls, implName, p1);
      // System.out.printf("To Lamda Def %s %s.%s(%s)\n", ifaceRet, ifaceCls, ifaceName, p2);
      // System.out.printf("invokedType = %s\n", invokedType);
      // System.out.printf("samMethodType = %s\n", samMethodType);
      // System.out.printf("instMethodType = %s\n", instMethodType);

      // Java 8 reference for the LambdaMetaFactory:
```

```
      // https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html#metafactory-java.la
ng.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-java.lang.invoke.MethodType-java.l
ang.invoke.MethodHandle-java.lang.invoke.MethodType-

      // The following code is mapped to the "3 phases" described in the class javadoc for
      // LambdaMetafactory.

      // Parameter descriptions from the LambdaMetafactory.metafactory() Javadoc:
      // caller        - Represents a lookup context with the accessibility privileges of the caller.
      //                 When used with invokedynamic, this is stacked automatically by the VM.
      // invokedName   - The name of the method to implement. When used with invokedynamic, this is
      //                 provided by the NameAndType of the InvokeDynamic structure and is stacked
      //                 automatically by the VM.
      // invokedType - The expected signature of the CallSite. The parameter types represent the types of capt
ure variables; the return type is the interface to implement. When used with invokedynamic, this is provided b
y the NameAndType of the InvokeDynamic structure and is stacked automatically by the VM. In the event that the
 implementation method is an instance method and this signature has any parameters, the first parameter in the
 invocation signature must correspond to the receiver.
      // samMethodType - Signature and return type of method to be implemented by the function object.
      // implMethod - A direct method handle describing the implementation method which should be called (with
 suitable adaptation of argument types, return types, and with captured arguments prepended to the invocation
arguments) at invocation time.
      // instantiatedMethodType - The signature and return type that should be enforced dynamically at invocat
ion time. This may be the same as samMethodType, or may be a specialization of it.

      // Return value description from the LambdaMetafactory.metafactory() Javadoc:
      // a CallSite whose target can be used to perform capture, generating instances of the interface named b
y invokedType

      // Phase 1: Linkage (we create a factory for the generation of lambdas that are linked to
      // the specific implementation method that is passed in AND those lambdas will implement
      // the given functional interface).
      CallSite site = LambdaMetafactory.metafactory(lookup,                // instance of MethodHandles.Looku
p which defined the caller's context
                                                    ifaceName,             // name of the invoked method (e.g
. for the functional interface Function the method name would be "apply")
                                                    invokedType,           // for static methods, there are n
o captured parameters (implicit references to the instance), this means the passed type is simply the target f
unctional interface
                                                                           // for instance methods, there is
a single captured parameter (the type of the class which implements the method that is being converted to a la
mbda), the passed type is defined by the target functional interface and the single captured parameter
                                                                           // since there is an existing meth
od passed in, there can be no other captured parameters since the method can only be referencing local data an
d member data, that is why there is only either zero (static) or one (instance) captured parameter
                                                    samMethodType,         // the functional interface signat
ure (return type, formal parameters); this is generic and does NOT represent the implementation method's speci
fic types
                                                    lookup.unreflect(impl), // this is the method handle that
has been converted from the implementation method that will be invoked by the lambda
                                                    instMethodType);       // the implementation signature (r
eturn type, formal parameters); this is NOT generic and represents the implementation method's specific types

      MethodHandle almost = site.getTarget();

      // Phase 2: Capture (only if there is an instance method, in that case we capture (bind) the instance)
      // NOTE: if the factory needs to be used may times and bound to more than one instance, then it can be
      //       returned and this phase could be implemented separately.
      return (instance == null) ? (T) almost.invoke() : (T) almost.bindTo(instance).invoke();
   }
```

Some considerations and things that need investigation:

- All locations where we use a Method instance, including locations where we call Utils.invoke(), can potentially be switched to this new approach.
- One new thing needed to make this work is to have access to the functional interface needed to call the method. This is potentially something new that we would need in each location.
- The test code I've attached shows examples of various getter and setter signatures. I'm not sure they will be used since Eric's idea for #6939 will be a better approach for the DMO implementation classes.
- I have not measured the cost of the conversion, but if the same lambda can be used many thousands of times then I'm hoping this will amortize well.
- In the current design of the lambdaFromMethod() helper, both phases 1 and 2 of the LambdaMetafactory are done at the same time. This means that any instance associated with the lambda is captured at the time of the call to the lambdaFromMethod() helper. This code can

actually be split so that phase 1 (creation of the specific lambda factory) is split from phase 2 (the capturing/binding of the instance). That could be used to save some amount of cost in the case where the same method is bound to many different instances.
- I would expect that the caller will handle caching for the lambda. There is no need for the result to be cached inside Utils.invoke(). Is the map lookup making the Utils.invoke() so much slower than the direct ReflectASM usage?
- In the lambdaFromMethod() helper, it is not clear if we can reuse the same lookup instance for all contexts. I haven't looked into that.

This change has the potential to be a significant improvement. It can be used much wider than just the places that use Utils.invoke(). To fully take advantage of it, I suspect much better caching at the caller will be needed.

Please review, comment and post any questions or thoughts.

#8 - 12/08/2022 08:02 AM - Greg Shah

I'm especially excited to see how this improves the ControlFlowOps usage since we could eliminate the dispatching cost of reflection and we already have a good way to cache the lambda at the caller.

#9 - 12/08/2022 10:03 AM - Constantin Asofiei

Greg Shah wrote:

> I'm especially excited to see how this improves the ControlFlowOps usage since we could eliminate the dispatching cost of reflection and we already have a good way to cache the lambda at the caller.

I'm testing ControlFlowOps now. But for other cases, the main issue with ReflectASM is the map lookup. For this, we need to change the FWD proxy impl to send the lambda beside the Method to the handler.

#10 - 12/08/2022 10:13 AM - Greg Shah

> For this, we need to change the FWD proxy impl to send the lambda beside the Method to the handler.

Eric's idea in #6939 is to eliminate the invocation handler in the proxy. Instead, a hook (or probably hooks) will be defined which can handle the getter/setter needs while returning back enough state to inform the generated bytecode of if/how to call the actual getter/setter. The idea is to have the actual call to the backing method be hard coded in the bytecode. This would be the fastest possible option and I think we should move ahead with it.

For non-proxy cases, your suggestion is correct. We'll also need to cache the result in the call site to try to eliminate the map lookup problem.

#11 - 12/08/2022 11:20 AM - Constantin Asofiei

Greg, for ControlFlowOps, this lambda method approach can't be easily used - because we will have to generate 'on the fly' the functional interface for each case.

#12 - 12/08/2022 11:24 AM - Greg Shah

We can generate Java code to register lambdas with CFO during an initializer. It could be an array where each element is a pair of lambda + name (internal entry or "execute").

#13 - 12/08/2022 12:41 PM - Constantin Asofiei

Greg Shah wrote:

> We can generate Java code to register lambdas with CFO during an initializer. It could be an array where each element is a pair of lambda + name (internal entry or "execute").

After more thought, I don't see how this can work; ControlFlowOps doesn't know the signature of what it will call. How will it pass the parameters?

#14 - 12/08/2022 12:52 PM - Constantin Asofiei

Constantin Asofiei wrote:

> Greg Shah wrote:
>
>> We can generate Java code to register lambdas with CFO during an initializer. It could be an array where each element is a pair of lambda + name (internal entry or "execute").
>
> After more thought, I don't see how this can work; ControlFlowOps doesn't know the signature of what it will call. How will it pass the parameters?

What I mean here, for a java.util.function.Function.apply(subscript) call, this can't be emulated in ControlFlowOps, as it needs to pass a 'generic' array of arguments.

#15 - 12/08/2022 01:41 PM - Constantin Asofiei

I've tested something like this (for my original MthdTest):

```
    FuncIface l = (arg) -> { testMe((String) arg[0]); return null; };
    n1 = System.nanoTime();
    for (int i = 0; i < n; i++)
    {
       l.get("bla");
    }
    n2 = System.nanoTime();
    System.out.println("Lambda call: " + (n2 - n1) / 1000000d);
```

and a:

```
  @FunctionalInterface
  public interface FuncIface
  {
     public Object get(Object... args);
  }
```

and the result on devsrv01 is:

```
ReflectASM: 241.712133
Utils.invoke: 217.611645
Method.invoke: 32.699907
Direct call: 4.82638
Lambda call: 8.044394
```

So this is (almost) on the same term as a direct call. This can be leveraged in the proxy generated bytecode, so that the invocation handler receives this general-purpose lambda, where it passes an array of arguments, and the proxy bytecode takes care (in the lambda's body) to put the parameters in place for the generated method call.

This removes ReflectASM from the picture for Utils.invoke and allows the invocation handler to work with only this lambda expression for the call, although the Method will be needed to be passed, to check the target's details (name, signature, etc).

The example above is only for static methods, for instance methods the first argument will need to be the target instance on which the call is performed. Here the problem will be the private methods, but in this case we can always fallback to Method.invoke for the call.

#16 - 12/08/2022 01:52 PM - Greg Shah

> How will it pass the parameters?

We can handle this with a bit of "standardization" in the registration or via adapter methods that get emitted in the converted code.

1. All lambdas passed would use:
   - Function for anything that is non-void return and the single parameter would be an array of BDT.
   - Consumer for void methods and the single parameter would be an array of BDT.
2. We would use one of two techniques:
   - Emit a wrapper lambda in the registration which has inline code to handle the adapting of the array elements to specific parameters (including casting to the write type); OR
   - We emit an extra adapter method that does this same thing as extra code in the class and then we just pass the lambda literal for the adapter method in the registration all.
3. Both versions would need an initializer block that does the registration.

The inline form might look like this:

```
private EntryPoint[] _dispatch_table_ = new EntryPoint[NUM_ENTRIES];

{
   _dispatch_table_[0] = new ExternalProcEntry("execute", this::execute);
   _dispatch_table_[1] = new InternalProcEntry("some-internal-proc", (args) -> someInternalProc((character)args[0], (integer)args[1])));
   _dispatch_table_[2] = new FunctionEntry("some-func", (args) -> return someFunc((decimal)args[0]));

   ControlFlowOps.registerEntryPoints(_dispatch_table_);
}
```

The inline form is probably the best approach. I'm not sure if the code above is exactly right, but hopefully the idea is clear.

The degenerate case of 0 parameters could be handled directly without adapters existing functional interfaces.

#17 - 12/08/2022 02:42 PM - Constantin Asofiei

Greg, the 8 million iterations chosen for the MthdTest was because this is the number of Utils.invoke for a large applications. In comparison, ControlFlowOps is called only ~50k times. Both cases can and need to be improved, but the solutions (although similar) will be different.

#18 - 12/08/2022 03:13 PM - Greg Shah

Yes. I assume the proxy case is being worked on in #6939.

#19 - 12/14/2022 01:26 PM - Constantin Asofiei

The lambdas will created like this:

```
public class RunPerfLambda
{
   private static final Map<Method, BiFunction<Object, Object[], Object>> lambdas = new HashMap<>();

   public static Map<Method, BiFunction<Object, Object[], Object>> getLambdas()
   {
      return lambdas;
   }

   static
   {
      Method m;
      try
      {
         m = Runperf.class.getDeclaredMethod("execute");
         lambdas.put(m, (ref, args) -> { ((Runperf) ref).execute(); return null; });

         m = Runperf.class.getDeclaredMethod("proc0", integer.class);
         lambdas.put(m, (ref, args) -> { ((Runperf) ref).proc0((integer) args[0]); return null; });
      }
      catch (Throwable t)
      {
         throw new RuntimeException(t);
      }
   }
}
```

This will be generated on-the-fly, when the Java class is first accessed, and the lambda references cached at the InternalEntry - from here, it gets propagated wherever needed in ControlFlowOps.

I'm working on the bytecode for this.

#20 - 12/19/2022 09:08 AM - Greg Shah

From Constantin:

> More with the the lambda for ControlFlowOps, I think the lambda performance is directly proportional with the number of lambdas created in the JVM - this may explain the slowdown I see, I need to capture some heaps for ETF and see how many java.lang.invoke lambda-related instances are created. In a standalone test, where I created 10000 classes, each with 300 lambdas (3 mil lambdas in total), even if 3 lambdas from all of these were being used, the performance decreased a lot (there may be some internal storage which affects the INVOKEDYNAMIC instruction at the JVM level).

#21 - 12/19/2022 09:08 AM - Greg Shah

> the performance decreased a lot (there may be some internal storage which affects the INVOKEDYNAMIC instruction at the JVM level)

It seems that when they need to be dynamically built in-memory, there are serious costs. I think that the JVM handles the hard coded lambdas in Java source code in a much better way. Consider how many lambdas we emit in the converted source code today and we don't have this issue (as far as I know).

If the theories are correct, then we should emit the adapter lambdas into the converted source code. Presumably this would eliminate the extra overhead. It is not as nice but fast is very important. We can even emit them at the end of the class so that they are less noticeable.

Then we just need an initializer that registers the lambdas and we should be able to avoid the issue.

#22 - 12/20/2022 04:40 AM - Constantin Asofiei

I've tested with a 'static bytecode' (i.e. 10k .java files each with 300 lambdas, checked the disassembled .class and there are 300 synthetic lambda methods in each class, so 3 million lambda in total) - Java behaves the same as the runtime-generated bytecode, there is a slowdown.

More, I had to move the lambdas from the 'compressed class space' to the heap (as this exceeded the max 3g allowed for the lambda space), -XX:-UseCompressedClassPointers -XX:-UseCompressedOops.

This fact that static bytecode behaves the same as runtime-generated bytecode is expected, as the bytecode for these runtime classes was built using as a 'template' a .class from a .java with the same structure.

OTOH, in ETF there are only ~100k created lambdas, after a full run. So my 3 million lambda test is on the edge of a stress-testing lambdas.

I'll work on doing some tests on a different machine and compare those. In the mean time, I'll prepare the changes and attach them as a patch.

#23 - 12/20/2022 02:19 PM - Constantin Asofiei

- File 6819_20221220a.patch added

Attached patch is built on top of #6939 patch. Eric, please review and use this one for your tests.

#24 - 12/21/2022 09:38 AM - Constantin Asofiei

Constantin Asofiei wrote:

> Attached patch is built on top of #6939 patch. Eric, please review and use this one for your tests.

This is built on top of 6129b/14341.

#25 - 12/23/2022 05:45 AM - Constantin Asofiei

- File 6819_20221223_6129b_14341.patch added

Constantin Asofiei wrote:

> Constantin Asofiei wrote:
>
>> Attached patch is built on top of #6939 patch. Eric, please review and use this one for your tests.
>
> This is built on top of 6129b/14341.

The correct 6129b patch is attached. This is the equivalent for the one #6939-16, for 3821c.

#26 - 01/06/2023 10:22 AM - Greg Shah

Code Review 6819_20221223_6129b_14341.patch

I have no objections.

#27 - 01/06/2023 10:22 AM - Greg Shah

Eric: Are you OK with the proxy and persistence changes?

#28 - 01/06/2023 10:44 AM - Greg Shah

The dynamic generation of the lambdas for an external procedure is only done the first time the external proc is loaded, right?

Did you compare the performance of the static (conversion generated) lambdas approach from #6819-16 with the dynamic load-time in-memory lambda assembly scenario that is implemented in the patch? Does the generation time amortize well? It seems like an expensive process.

#29 - 01/06/2023 02:51 PM - Constantin Asofiei

Greg Shah wrote:

> The dynamic generation of the lambdas for an external procedure is only done the first time the external proc is loaded, right?

Is done only once, correct.

> Did you compare the performance of the static (conversion generated) lambdas approach from #6819-16 with the dynamic load-time in-memory lambda assembly scenario that is implemented in the patch? Does the generation time amortize well? It seems like an expensive process.

Yes, I did - the dynamic bytecode I generate is the same as for the static case I tested. There are no differences, both static and dynamic behaves the same when stress testing - see #6819-22

#30 - 01/06/2023 03:05 PM - Greg Shah

I understand there is no performance difference in lambdas themselves. I'm wondering about the call to LambdaFactory.buildLambdas(). That has a cost as compared to the static code generation which would just register existing lambdas.

#31 - 01/09/2023 08:57 AM - Constantin Asofiei

Greg Shah wrote:

> I understand there is no performance difference in lambdas themselves. I'm wondering about the call to LambdaFactory.buildLambdas(). That has a cost as compared to the static code generation which would just register existing lambdas.

This is called only once and cached at the InternalEntry. This is not something which would interfere ~~before~~ after a server warm-up.

#32 - 01/20/2023 09:27 AM - Constantin Asofiei

- % Done changed from 0 to 50

A resume of what was done at this task:

Experiments were done using lambda expression instead of Java reflection or ReflectASM. Although standalone tests proved that lambda expression are faster, when having 100k's or millions of registered lambdas, the lambda invocation will behave slower than reflection. This may be because of how JVM keeps the lambda registry; further investigation is required to understand why the JVM shows a slowdown in real applications.

There is no difference between a conversion-time generated lambda in a .java file and a runtime-generated .class bytecode defining these lambdas, in a 'lambda execution' POV. Experiments were done having 10k classes defining 100s of lambdas each, and loading all these classes into JVM - even if one single lambda is being invoked, the number of loaded lambdas affects performance.

#33 - 02/12/2023 03:14 AM - Constantin Asofiei

- File lambda_perf.png added

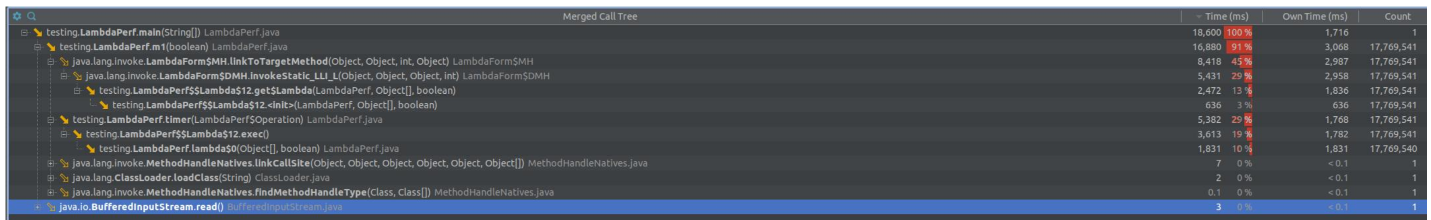
I've did more testing and profiling of lambda calls. Even a simple lambda call as we have in ContextLocal.get(boolean):

```
public final T get(boolean create)
{
   Object[] res = new Object[1];

   GET.timer(() -> res[0] = getImpl(create));

   return (T) res[0];
}
```

when executing 18 million times is 450ms vs 4ms on devsrv01. I've profiled the equivalent code in a standalone program, and the tracing is the same as the one for a large application profiling.



Merged Call Tree	Time (ms)		Own Time (ms)	Count
testing.LambdaPerf.main(String[]) LambdaPerf.java	18,600	100 %	1,716	1
testing.LambdaPerf.m1(boolean) LambdaPerf.java	16,880	91 %	3,068	17,769,541
java.lang.invoke.LambdaForm$MH.linkToTargetMethod(Object, Object, int, Object) LambdaForm$MH	8,418	45 %	2,987	17,769,541
java.lang.invoke.LambdaForm$DMH.invokeStatic_LLI_L(Object, Object, Object, int) LambdaForm$DMH	5,431	29 %	2,958	17,769,541
testing.LambdaPerf$$Lambda$12.get$Lambda(LambdaPerf, Object[], boolean)	2,472	13 %	1,836	17,769,541
testing.LambdaPerf$$Lambda$12.<init>(LambdaPerf, Object[], boolean)	636	3 %	636	17,769,541
testing.LambdaPerf.timer(LambdaPerf$Operation) LambdaPerf.java	5,382	29 %	1,768	17,769,541
testing.LambdaPerf$$Lambda$12.exec()	3,613	19 %	1,782	17,769,541
testing.LambdaPerf.lambda$0(Object[], boolean) LambdaPerf.java	1,831	10 %	1,831	17,769,540
java.lang.invoke.MethodHandleNatives.linkCallSite(Object, Object, Object, Object, Object, Object[]) MethodHandleNatives.java	7	0 %	< 0.1	1
java.lang.ClassLoader.loadClass(String) ClassLoader.java	2	0 %	< 0.1	1
java.lang.invoke.MethodHandleNatives.findMethodHandleType(Class, Class[]) MethodHandleNatives.java	0.1	0 %	< 0.1	1
java.io.BufferedInputStream.read() BufferedInputStream.java	3	0 %	< 0.1	1

I think we need to add a JVM flag to enable or disable the JMX code, so for production mode it will go via the direct call and not the lambda call.

Greg: we need to consider if we want to make it a policy to discourage Java lambda, the Java stream API, and even the : 'for each', especially in the core runtime. This article is a good read: https://www.beyondjava.net/performance-java-8-lambdas

I've even tested with Java 11 and the performance impact is similar. I've also tried to leave the ContextLocal.get JMX code in place and replace the lambda with a anon class instance, but the impact of creating some 16mm instances and garbage-collect these is affecting performance.

#34 - 02/13/2023 08:04 AM - Greg Shah

> I think we need to add a JVM flag to enable or disable the JMX code, so for production mode it will go via the direct call and not the lambda call.

Do you mean a FWD-specific property that we would query and then conditionally bypass the JMX calls? Or something else?

#35 - 02/13/2023 08:06 AM - Constantin Asofiei

Greg Shah wrote:

> > I think we need to add a JVM flag to enable or disable the JMX code, so for production mode it will go via the direct call and not the lambda call.
>
> Do you mean a FWD-specific property that we would query and then conditionally bypass the JMX calls? Or something else?

Yes, a property via -D argument at the java command.

#36 - 02/13/2023 08:17 AM - Greg Shah

How many JMX instrumentation locations are there? I'd like to run some non-trivial application code with and without JMX enabled and know the difference.

We went out of our way to try to make the JMX code transparent from a performance perspective, but even so it seems to add up. If we have to go this route, then it would suggest that to use JMX will require a server restart and some small but measurable performance hit. Mosts importantly, JMX would not be usable for normal production monitoring which is at least half of the idea of JMX.

The alternative is to rewrite each of the JMX locations to expand the code in place and use all direct/fast access. Yuck. That would allow us to eliminate any lambda usage at the cost of exploding the code out everywhere. Not nice, but it should be performant.

#37 - 02/13/2023 08:39 AM - Constantin Asofiei

- File LambdaPerf.java added

Try attached testcase in command-line. On devsrv01, the times are these:

```
Lambda call:   475.114187
Runnable call: 178.136995
Direct call:   6.038513
Direct call with throw: 5.213002
```

In the profiler of a large customer app, these stand out:

- ContextLocal.get() - 16.5mm calls, related to JMX, tested with the JMX timer removed
- ScopedDictionary.addScope(Object) - 4.2mm calls, not related to JMX, solved by replacing the lambda
- handle.set(WrappedResource) - 1mm calls, not solved, the checkUndoable(() -> new handle(value)); line
- TableMapper$LegacyFieldInfo.getMutableInfo() - 1.15mm calls, tested by creating a Supplier instance instead of lambda, but I'm not sure yet about this sollution.
- UnclosablePreparedStatement.exec() - 125k, related to JMX, not solved

From these, only ContextLocal.get() is the major impact (with the calls from the TRPL dynamic conversion, from AstSymbolResolver.getResolver()). Short term, we can remove the JMX timer for ContextLocal.get().

#38 - 02/13/2023 08:42 AM - Constantin Asofiei

Greg Shah wrote:

> How many JMX instrumentation locations are there? I'd like to run some non-trivial application code with and without JMX enabled and know the difference.

There are 47 instances of NanoTimer, which is the only one where a lambda is required.

#39 - 02/15/2023 09:04 AM - Alexandru Lungu

I've run LambdaPerf.java and I've got similar results. On my computer however, I get Lambda Call time (132.047478) just a bit slower than Runnable Call (125.866193) - so there is no such big deal between these two
I attempted some new test-case to try out a solution which still keeps the JMX decoupled:

- I changed Operation to return the result, without the need of a separate res array. The timer function takes as arguments a parameterized lambda and its parameters (tested with only one parameter). The time went to 11.068474:

  ```
  Object res = timer((Operation) (cr) -> getImpl((boolean) cr), create);
  ```

- I extracted the lambda into a class property and run the test from above. The time is 4.962356:

  ```
  Operation op = (Operation) (cr) -> getImpl((boolean) cr); [...] Object res = timer(op, create);
  ```

We can generalize the technique above to use a variable number of parameters. Something like Object exec(Object... args) for Operation and Object timer(Operation op, Object... args) for timer. In fact, I tested this and for one parameter at least I have the same time results:

```
Object res = timer((Operation) (cr) -> getImpl((boolean) cr[0]), create);
```

Unless my times are due to a Java compilation optimization, cutting down the code altogether, I think it is worth attempting such solution.

#40 - 02/15/2023 09:08 AM - Constantin Asofiei

Alexandru Lungu wrote:

> Unless my times are due to a Java compilation optimization, cutting down the code altogether, I think it is worth attempting such solution.

Please make the changes on top of 7026a and post the patch here.

#41 - 02/15/2023 10:15 AM - Constantin Asofiei

Greg, if Alexandru's idea is proven successful, then I think the bytecode can be changed from:

```
lambdas.put(m, (ref, args) -> { ((Runperf) ref).execute(); return null; });
```

to:

```
private static final Function<Object, Object[] LAMBDA_1 = (ref, args) -> { ((Runperf) ref).execute(); return n
ull; };
...
lambdas.put(m, LAMBDA_1);
```

I need to do some testing and profiling myself, but I think the problem may be with the lambda linking (as currently the linking is done on each call), and not with the actual invocation.

#42 - 02/15/2023 11:11 AM - Constantin Asofiei

I've tried this idea with something like this in LambdaPerf:

```
   private final Operation op = (Operation) () -> { getImpl(false); };

   public void m1c(boolean create)
   {
      timer(op);
   }
```

and the time on devsrv01 is 79ms vs 7ms direct call. The same on my machine, 34ms vs 2.74ms.

Now, after I looked again at the generated lambda code for this task, the lambdas are already 'kind of static' - the lambda map is populated only once. So it shouldn't make any difference if the lambda map is populated from a static field or a inlined lambda - but I will test.

But, I've profiled the scenario with the inlined lambda vs the static lambda, and:

- for the static case, the Eden space is not doing anything - no new objects allocated, etc. The app is just executing the lambda.
- for the inlined lambda case (what is in LambdaPerf.java), the Eden space is doing a lot of work, and so is the garbage collector.

My conclusion at this time:

- Alexandru, go ahead with the changes to move to static lambda definitions, and not inlined, as inlined lambdas seem to be the culprit.
- I'll test the proxy lambda generated code and see how the profiler reports the Eden space for them.

#43 - 02/15/2023 11:19 AM - Constantin Asofiei

Actually, the inlined lambda Eden space allocations is because of the Object[] res = new Object[1]; array. Alexandru, I expect the same problem will be if you will pass the arguments as a vararg - this is a 'vararg' only in name, the bytecode will allocated a Object[] array for it, on each call. This will be expensive in terms of GC.

My static lambda case had no arguments and that is why the Eden space wasn't showing anything.

You can use VisualVM 2.1.4 and the Visual GC plugin (use Tools -> Plugins to install it), to check the Eden space.

#44 - 02/15/2023 11:39 AM - Constantin Asofiei

Constantin Asofiei wrote:

> Now, after I looked again at the generated lambda code for this task, the lambdas are already 'kind of static' - the lambda map is populated only once. So it shouldn't make any difference if the lambda map is populated from a static field or a inlined lambda - but I will test.

The lambdas are already static for ControlFlowOps usage, but not for the i.e. DMO proxy usage - in this case, they are statically linked.

But after further insight, the actual slowdown seems to be from the array reference from within the lambda. I'm curious how the Operation.exec returning a value will impact this. OTOH, if you have the 'lambda' like this:

```
   private final Operation op2 = new Operation()
   {
      @Override
      public void exec() throws Exception
      {
         getImpl(false);
      }
   };
```

instead of this:

```
private final Operation op = (Operation) () -> { getImpl(false); };
```

this is reported almost as fast as a direct call.

So, it seems that another idea would be to not use lambdas, but anon singleton instances, where possible.

#45 - 02/15/2023 11:46 AM - Alexandru Lungu

- File stateless_lambda.patch added

I've done a patch, but in a bit of a rush.

There are several complex JMX which need rethinking, as there are > 5 bound variables. Also, the SQL logger needs a bit of rethinking in terms of lambda usage. The patch has only trivial lambdas fixed.
I did not test it yet with any large scale scenario yet. I will need to do some analysis on my own first thing tomorrow morning.
From the "statically extract lambda as class member" point of view, I think it is really bad to have chunks of logical code in the top of the class zone. Some methods will look like: timer(op, buf, dmo, meta) (where op is just an out-of-the-context top-level lambda member field of the class) and will decrease visibility of code. If really needed for performance, maybe we can compromise on this.

#46 - 02/15/2023 11:51 AM - Constantin Asofiei

Alexandru Lungu wrote:

> From the "statically extract lambda as class member" point of view...

My analysis seems to have been skewed by the Object[] res = new Object[1]; and the res[0] = getImpl() usage from the lambda. After I removed the 'returned value' for all cases, the singleton lambda call and the inlined lambda call both have the same time.

But, the 'singleton runnable' case (where the lambda doesn't reference any local variables or arguments), is almost as fast as the direct call.

I'll look at the patch, but I don't think allocated a vararg is a good approach. I'll test with more than one reference at the lambda (local var and argument) and see how this behaves.

#47 - 02/16/2023 05:09 AM - Alexandru Lungu

Constantin Asofiei wrote:

> I'll look at the patch, but I don't think allocated a vararg is a good approach. I'll test with more than one reference at the lambda (local var and argument) and see how this behaves.

I was thinking at same thing while doing the patch. Maybe we can switch to an overloaded timer method approach, with 0/1/2/3/4 parameters which hold the actual references from the lambda? As I was surfing through the JMX, there are only some lambdas with more than 4 dependencies, that should be refactored anyways. Using the stack to move the parameters around should be definitely faster, than having a heap object array to hold them (+ GC overhead).

> My analysis seems to have been skewed by the Object[] res = new Object[1]; and the res[0] = getImpl() usage from the lambda. After I removed the 'returned value' for all cases, the singleton lambda call and the inlined lambda call both have the same time.

I've done some extensive testing and I agree that the Object[] res = new Object[1]; was actually skewing the tests. I guess we shouldn't expect a ~130ms to ~5ms time decrease now. I extracted Object[] res as a class member and I've got ~30ms with an inlined lambda with references to variables and arguments (basically the exact test from LambdaPerf with one single res allocation). So at best, we can expect a ~24ms time improvement from avoiding lambda references?

> But, the 'singleton runnable' case (where the lambda doesn't reference any local variables or arguments), is almost as fast as the direct call.

It was hard for me to notice at first that this is sometimes implicitly bound to the lambda call (e.g. in LambdaPerf, getImpl forces the lambda to reference this).

```
Object res = timer((Operation2) (target, cr) -> ((LambdaPerf) target).getImpl((boolean) cr), this, create);
```

This is as fast as direct access, but note that **the lambda is still inlined**! Also, I use Operation2 without vargs. On a more general case, each reference linked to the lambda is bringing its own overhead (including ret[0], create and this).

> I'll look at the patch, but I don't think allocated a vararg is a good approach. I'll test with more than one reference at the lambda (local var and argument) and see how this behaves.

The patch doesn't use the "this as argument" optimization and is relying on the inconsistent results of Object[] res = new Object[1];. Therefore, **the patch should be redone**.

My point here is that ~24ms don't look like a benefit considering that we need to "butcher" the instrumentation code: creating Operation0, Operation1, Operation2, etc., enforce an instrumentation policy of avoiding local references, the profiled code is no longer going to link and type-check the references, but needs to explicitly cast (unless we use generics).

I think we should rather target the timers which actually use Object[] res = new Object[1]; and are called thousands of times.. For GET.timer(() -> res[0] = getImpl(create));, the concern is not necessarily the variable/argument referencing, but the creation of res[0] before-hand. In this case, using return GET.timer(() -> getImpl(create)); instead should be almost enough. For exceptional cases (with million of calls), I guess we can go the extra mile and avoid variable/argument referencing and use return GET.timer((target, arg) -> ((...) target).getImpl((...) arg), this, create);, but it shouldn't be the common case.

I suggest targeting #6819-37 with all we got (remove variable/argument lambda referencing) and just touch the other JMX timers to avoid Object[] res = new Object[1] and other array constructs dedicated to lambda profiling (if any exists).

#48 - 02/16/2023 05:16 AM - Constantin Asofiei

Alexandru, please create a patch to fix the ContextLocal.get (the Object[] res) first. This is something we can solve easily, to add a return value to the timer() method.

#49 - 02/16/2023 05:28 AM - Alexandru Lungu

- File context_local_jmx.patch added

Done!

#50 - 02/16/2023 05:32 AM - Constantin Asofiei

Alexandru Lungu wrote:

> Done!

Great! Make sure to add history numbers to any changed file (just a reminder, history numbers are added once per file per branch). And NanoTimer is missing history entry. Please commit to 7026a

#51 - 02/16/2023 05:48 AM - Constantin Asofiei

Status of the #6819-37 issues:

- ContextLocal.get() - 16.5mm calls, related to JMX, tested with the JMX timer removed - **solved** by Alexandru's patch, need to run testing with it
- ScopedDictionary.addScope(Object) - 4.2mm calls, not related to JMX, **solved by replacing the lambda**, change is in 7026a

- handle.set(WrappedResource) - 1mm calls, ~~not solved~~, the checkUndoable(() -> new handle(value)); line - **solved** by calling checkUndoable only for undoable instances, change is in 7026a
- TableMapper$LegacyFieldInfo.getMutableInfo() - 1.15mm calls, tested by creating a Supplier instance instead of lambda, but I'm not sure yet about this solution **not solved**
- UnclosablePreparedStatement.exec() - 125k, related to JMX, ~~**not solved**~~ solved in 7026a/14495

#52 - 02/16/2023 06:08 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Alexandru Lungu wrote:
>
> > Done!
>
> Great! Make sure to add history numbers to any changed file (just a reminder, history numbers are added once per file per branch). And NanoTimer is missing history entry. Please commit to 7026a

Committed 7026/rev. 14495 including the patch from #6819-49. I also applied the same timerWithReturn to optimize UnclosablePreparedStatement.exec(). Please review.

#53 - 02/16/2023 06:34 AM - Constantin Asofiei

Review for 7026/rev. 14495. In NanoTimer, UnclosablePreparedStatement, ContextLocal, please add history numbers, instead of:

```
**     AL2 20230216 Added timer method able to return the value of the underlying instrumented code.
```

it needs to be:

```
** 002 AL2 20230216 Added timer method able to return the value of the underlying instrumented code.
```

as this is the first change of this file in this branch.

#54 - 02/16/2023 07:36 AM - Alexandru Lungu

Constantin Asofiei wrote:

> Review for 7026/rev. 14495. In NanoTimer, UnclosablePreparedStatement, ContextLocal, please add history numbers, instead of:
> [...]
> it needs to be:
> [...]
> as this is the first change of this file in this branch.

Done in 7026a/14496. I didn't get what you meant by "history numbers" at first. Guess I was stuck on committing on a single branch for some time now.

#55 - 02/17/2023 02:27 PM - Eric Faulhaber

Constantin Asofiei wrote:

> Greg: we need to consider if we want to make it a policy to discourage Java lambda, the Java stream API, and even the : 'for each', especially in the core runtime. This article is a good read: https://www.beyondjava.net/performance-java-8-lambdas

+1, at least for the stream API and the "for each" loop in core runtime infrastructure code.

I use lambdas pretty heavily with Map collections, to replace get/put sequences with computeIfAbsent. I do this primarily to avoid the performance hit of multiple calls to lookup a value in the map. But, could the use of lambdas here be doing more harm than good? I haven't seen anything published which address this point specifically.

#56 - 02/24/2023 06:30 AM - Constantin Asofiei

I've changed the original #6819-3 test so that a new Object[] (for the varargs) is not created on each call (so a single instance is used). Now, the times are these on devsrv01:

- ReflectASM: 120.333418 - this is pure MethodAccess.invoke with a method index
- Method.invoke: 87.568592 - this is pure java.lang.Method.invoke
- Utils.invoke: 632.616997 - this is using ReflectASM via Utils.invoke, with the map lookup overhead
- Direct call: 6.265417 - plain method call
- Lambda call: 7.821601 - plain singleton lambda call, no capturing

On my machine, the times are:

```
ReflectASM: 13.152562
Method.invoke: 27.151114
Utils.invoke: 192.592094
Direct call: 3.711335
```

```
Lambda call: 4.763449
```

So, going back to original test, the times were skewed because of the allocation of the Object[] vararg instance.

With this in mind, I think it shouldn't take too long to change the invocation handler to receive the info for ReflectASM, MethodAccess access, int methodIndex, besides the target method, while ensuring that the MethodAccess instance is a constant in the proxy code. In absolute terms, there is a small difference for 18mm calls, and it seems to be worth it to check how ReflectASM behaves, too; but again, the test times may be skewed on how the hot-spot compiler decides to compile code to native.

The current iteration of the lambda proxy attempts is to use a BiFunction[] la array like we do for the Method[] ma, with the la array populated with static defined lambdas, in a separate dynamically generated class (one per each proxied class, same as ReflectASM does for MethodAccess). Even with this, the time does not improve.

For example, disassembling this dynamically generated lambda:

```
public static final BiFunction LAMBDA$2 = (var0, var1) -> { var0._errorFlags((Integer)var1[0]); return null; }
;
```

results in this bytecode being invoked:

```
    private static synthetic lambda$2(com.goldencode.p2j.persist.TempTableRecord arg0, java.lang.Object[] arg1
) { //(Lcom/goldencode/p2j/persist/TempTableRecord;[Ljava/lang/Object;)Ljava/lang/Object;
             aload0 // reference to arg0
             aload1
             iconst_0
             aaload
             checkcast java/lang/Integer
             invokeinterface com/goldencode/p2j/persist/TempTableRecord._errorFlags(Ljava/lang/Integer;)V
             aconst_null
             areturn
     }
```

Keep in mind that previously arg0 for ByFunction was java.lang.Object, which required another checkcast instruction. But checkcast is still required for the arguments, plus all the loading from the array.

This generated lambda in turns ends up being passed to this proxied method:

```
   private static final BiFunction[] la;
   ....

   public void _originRowid(Long var1) {
      this.h.invoke(this, ma[6], la[6], new Object[]{var1});
   }
```

From what I can tell, there are no new Object[] array allocations for the arguments, this is done only once at the original proxied call, and passed until the lambda is reached.

#57 - 02/24/2023 10:03 AM - Greg Shah

> With this in mind, I think it shouldn't take too long to change the invocation handler to receive the info for ReflectASM, MethodAccess access, int methodIndex, besides the target method, while ensuring that the MethodAccess instance is a constant in the proxy code.

Is this proposal meant to be a temporary step while we work out the lambda approach?

#58 - 02/24/2023 10:06 AM - Constantin Asofiei

Greg Shah wrote:

> > With this in mind, I think it shouldn't take too long to change the invocation handler to receive the info for ReflectASM, MethodAccess access, int methodIndex, besides the target method, while ensuring that the MethodAccess instance is a constant in the proxy code.
>
> Is this proposal meant to be a temporary step while we work out the lambda approach?

I want to see direct ReflectASM in action with a large customer app, and see how it behaves.

#59 - 05/10/2023 09:02 AM - Constantin Asofiei

Branch 6819a was created from trunk rev 14563. Rev 14564 contains the (MethodAccess, index) arguments at the proxy invocation handler .

Branch 6819b was created from trunk rev 14563. Rev 14564 contains the BiFunction argument at the proxy invocation handler. For lambda case, there is another possible approach: the lambda synthetic methods are (mostly) the same, for i.e. RecordBuffer and other internal usage. From some reading, it may be possible to re-use the lambda synthetic method from another class.

#60 - 05/11/2023 10:53 AM - Eric Faulhaber

- Related to Feature #7348: mutable proxy implementation added

#61 - 05/11/2023 12:43 PM - Constantin Asofiei

So, I've expanded 6819b to use 'public static' lambdas defined only once, in a distinct $__<type>Lambda class. The performance is still not OK, even though the number of lambdas has been drastically reduced (from a lambda def copy in each proxy, to a single lambda def).

I've been experimenting a little more with MethodHandle. The problem here is that the argument number varies, and invokeWithArguments is very-very-very-slow.

But I think I found something which allows the invoke with an Object[] array and brings the call to almost native call parity: MethodHandle.asSpreader(Object[].class, <number-of-arguments>); this allows calls like this:

- mh.invoke(args); for static calls
- mh.invoke(instance, args); for instance calls

where args is a Object[] array.

I think I can modify 6819a (which uses MethodHandle and index argument at the handler.invoke), to:

- pre-calculate and pre-process MethodHandle instances for all proxied methods, and keep them in a mha static array at the proxy.
- the handler.invoke receives a MethodHandle argument and uses either invoke(args) or invoke(instance, args), depending if instance is null or not. Now, the downside is this: the security access check is done **both** at where the lookup.unreflect is performed and where the call is performed! if the method is not visible in both of these places, then the call will fail; so:
  - at runtime, I think it may be possible to just directly call mh.invoke instead of doing it via Utils.invoke, and i.e. calls from p2j.persist.RecordBuffer should be OK. But this does not solve cases for RUN statement - in this case, only public methods will be ran via MethodHandle, the rest will default to Method.invoke.
  - at proxy generation time, the MethodHandle needs to be resolved in the proxy bytecode directly, so all methods can be resolved in the proxy's context.

There is some complexity for the bytecode generation, but I don't think it will take more than a couple of days.

#62 - 11/08/2023 03:30 AM - Constantin Asofiei

For a while, I've been looking through videos, blog posts and other info I could get my hands on, related to lambdas. I think the conclusion is this:

- call-sites with capturing lambdas will be resolve on each call, like you would instantiate an anonymous class
- call-sites with non-capturing lambdas - JVM does its best to keep the call-site's lambda as a singleton, but this singleton can still be garbage-collected. So, once the singleton is gone, it will again resolve the lambda. 6819b tried this exact scenario: use non-capturing lambdas which are supposed to be singletons, but they still get garbage collected.

I've tried to find some info if anything has changed with Java 11 or Java 17 related to how lambdas are resolved and garbage collected, but I couldn't find anything relevant.

I still have one idea to test, to use real singletons which never get garbage collected:

- create (similar to 6819b) i.e. a BufferImplCallSite class, which will have 'public static final' fields but, instead of the lambda, will hold an anonymous instance of BiFunction<Object, Object[], Object>. Its apply method will just do method invocation with the specified arguments, on the given reference.
- as these instances are referenced by class static fields, they should never get garbage collected.

#63 - 11/10/2023 09:50 AM - Alexandru Lungu

Constantin, I reached quite randomly in Utils.invoke and saw the double map access on METHOD_DATA and methodAccessIdx. Both store either Class or Method and yet are ConcurrentHashMap and HashMap. Can we make them into IdentityHashMap (and some concurrent counterpart) or is there some reflection shenanigans we do so that Identity is not OK to be used here?

#64 - 11/10/2023 09:56 AM - Constantin Asofiei

That code resolves the MethodAccess target for a java.lang.reflect.Method. Method instances are not singleton, so IdentityHashMap can't be used. But I think you are right - we can use a single ConcurrentHashMap<Method, MethodData> map, where MethodData holds the MethodAccess access instance and a Integer methodAccessIdx field - so a single map lookup is used.

The change should be simple, the only caveat is that when a Method is missing, then we need to lock on something (the java.lang.Class?) so only a single MethodAccess instance gets created; can you give it a try and see the performance impact?

#65 - 11/21/2023 01:56 PM - Greg Shah

Please post details of the results from your testing described in #6819-61 (6 months ago).

I understand you've been doing other testing recently. Please describe the details and results of that work.

#66 - 11/21/2023 02:43 PM - Constantin Asofiei

Currently there are three experimental branches:

- 6819c - uses singleton, byte-code generated, BiFunction inner classes
- 6819b - uses pure BiFunction lambdas (assumed singleton, generated in their own class like BufferImplLambdas)
- 6819a, which has two commits, both of these create fields at the Proxy class, and not singletons like at BufferImplLambdas in 6819b:
    - MethodAccess usage in rev 14564
    - MethodHandle usage in rev 14565

All these attempts proved unsuccessful: the timing was worse with ~2%.

The main explanation I think is because the invokedynamic JVM instruction relies on the JVM implementation, where the internal data structures are prone to garbage collection, and they are not 'once and done', even for non-capturing lambdas (which are supposed to be singletons).

We may want to revisit this once we move to Java 17, but until then, I'm out of ideas.

#67 - 11/22/2023 08:22 AM - Greg Shah

Please post the test harness code or whatever it is that you are using to create the failing case ("where I created 10000 classes, each with 300 lambdas (3 mil lambdas in total)").

What specific use cases would be using the modified code? Is it just the proxied DMOs?

#68 - 11/22/2023 08:56 AM - Constantin Asofiei

Greg Shah wrote:

> Please post the test harness code or whatever it is that you are using to create the failing case ("where I created 10000 classes, each with 300 lambdas (3 mil lambdas in total)").

I will.

> What specific use cases would be using the modified code?

RecordBuffer, TemporarBuffer, DataSet, ControlFlowOps are using the modified code.

#69 - 11/22/2023 03:02 PM - Greg Shah

> > What specific use cases would be using the modified code?
>
> RecordBuffer, TemporarBuffer, DataSet, ControlFlowOps are using the modified code.

Do we have metrics about the number of these classes (and the number of lambdas) that exist in projects for these classes?

Have you tested the various branches with real projects to see the results?

Files

mthdtest.zip	2.48 KB	10/28/2022	Constantin Asofiei
mthdtest_converted_lambda_approach_20221207.zip	10.1 KB	12/08/2022	Greg Shah
6819_20221220a.patch	43.5 KB	12/20/2022	Constantin Asofiei
6819_20221223_6129b_14341.patch	91.3 KB	12/23/2022	Constantin Asofiei
lambda_perf.png	127 KB	02/12/2023	Constantin Asofiei
LambdaPerf.java	2.17 KB	02/13/2023	Constantin Asofiei
stateless_lambda.patch	34.8 KB	02/15/2023	Alexandru Lungu
context_local_jmx.patch	6.55 KB	02/16/2023	Alexandru Lungu