Fast late-bound invocation through DynamicMethod delegates

By **Alessandro Febretti**, 11 Jul 2005

Introduction

Reflection is a feature that allows a program to find out type (and metadata) information about objects at run-time. Programs written in languages that support Reflection, like Java and the CLR languages family (C#, Visual Basic .NET etc.) can inspect types, obtain detailed information about class members, dynamically instantiate classes and invoke methods at run-time. The .NET Framework exposes its reflection services through the `System.Reflection` namespace. Late-bound invocation, for instance, can be achieved through the `Type.InvokeMember` and `MemberInfo.Invoke` methods. The use of Reflection, however, comes with a price. While some of its functions are pretty fast, some others, like late-bound invocation routines, are costly and if not used wisely could result in a major bottleneck inside your application.

Joel Pobar in his article Dodge Common Performance Pitfalls to Craft Speedy Applications offers a good overview over .NET Reflection performance, as well as describes a way to obtain fast late-bound invocation, the implementation of which is discussed in this article.

Hybrid Late-Bound to Early-Bound Invocation through DynamicMethod Delegates

The basic idea presented by Joel Polbar is to obtain a method handle through the common Reflection services, and then emit MSIL code to patch up a static call site for the method. This call site will be kept as simple as possible: no type safety / security checks will be performed, assuming that hybrid invocation is done only between fully trusted code entities. Generation of the MSIL call site will be done through a new feature of the .NET Framework 2.0, called Lightweight Code Generation (LGC). Hybrid invocation can be achieved also under .NET Framework 1.1 using the standard `Reflection.Emit` facilities, but at the price of a heavier implementation.

The DynamicMethod Class

Lightweight Code Generation features are exposed by a single new class inside the `System.Reflection.Emit` namespace: `DynamicMethod`. You can use this class to generate and execute a method at run time, without having to generate a dynamic assembly and a dynamic type to contain the method. Dynamic methods are the most efficient way to generate and execute small amounts of code.

```
public DynamicMethod(
    string name,
    Type returnType,
    Type[] parameterTypes,
    Type owner
);

public DynamicMethod(
    stringname,
    TypereturnType,
    Type[] parameterTypes,
    Module owner
);
```

These are two public constructors available for the class. As you can see, a dynamic method is logically associated with a module or with a type (`owner` parameter).

Code Overview

The entire implementation resides in a `static` method:

```
public static DynamicMethodDelegate DynamicMethodDelegateFactory.Create(
    MethodInfo method
);
```

This function takes a `MethodInfo` representing the method we want to invoke, and returns a `DynamicMethodDelegate` delegate:

```
public delegate object DynamicMethodDelegate(
    object instance,
    object[] args
);
```

The delegate syntax is the same as one of the `MethodInfo.Invoke` overloads: it takes an `object` instance (that can be null in the case of static methods) and an array of arguments, and returns a generic `object` instance.

Using the code is pretty straightforward:

```
MyClass instance = new MyClass();
MethodInfo myMethodInfo;
// [...] myMethodInfo = some method of MyClass

DynamicMethodDelegate myDelegate;

// Generate delegate for myMethodInfo
myDelegate = DynamicMethodDelegateFactory.Create(myMethodInfo);
```

```
// Invoke method through delegate.
object[] args = {/* method arguments here */};
object result = myDelegate(instance, args);
```

Implementation Details

The initial part of our method is quite self explaining:

```
/// <summary>
/// Generates a DynamicMethodDelegate delegate from a MethodInfo object.
/// </summary>
public static DynamicMethodDelegate Create(MethodInfo method)
{
    ParameterInfo[] parms = method.GetParameters();
    int numparams = parms.Length;

    Type[] _argTypes = { typeof(object), typeof(object[]) };

    // Create dynamic method and obtain its IL generator to
    // inject code.
    DynamicMethod dynam =
        new DynamicMethod(
        "",
        typeof(object),
        _argTypes,
        typeof(DynamicMethodDelegateFactory));
        ILGenerator il = dynam.GetILGenerator();

    /* [...IL GENERATION...] */

    return (DynamicMethodDelegate)
      dynam.CreateDelegate(typeof(DynamicMethodDelegate));
}
```

Let's now focus our attention on the IL generation code. Our IL call site will be divided into four sections:

- Argument count check
- Object instance push
- Argument layout
- Method call

Argument count check

```
// Define a label for succesfull argument count checking.
Label argsOK = il.DefineLabel();

// Check input argument count.
il.Emit(OpCodes.Ldarg_1);
il.Emit(OpCodes.Ldlen);
il.Emit(OpCodes.Ldc_I4, numparams);
il.Emit(OpCodes.Beq, argsOK);

// Argument count was wrong, throw TargetParameterCountException.
il.Emit(OpCodes.Newobj,
   typeof(TargetParameterCountException).GetConstructor(Type.EmptyTypes));
il.Emit(OpCodes.Throw);
```

```
// Mark IL with argsOK label.
il.MarkLabel(argsOK);
```

In this section we compare the length of the provided input arguments array with the number of parameters accepted by the method. On equality, we proceed to the next section (marked with the `argsOK` label), otherwise a `TargetParameterCountException` is thrown.

Object instance push

```
            // If method isn't static push target instance on top
            // of stack.
            if (!method.IsStatic)
            {
                // Argument 0 of dynamic method is target instance.
                il.Emit(OpCodes.Ldarg_0);
            }
```

Clear and simple. The function call MSIL opcodes (`call`, `calli`, `callvirt`) need the target object reference to be pushed onto the stack before the arguments. This obviously applies only in the case of non-static method calls.

Argument layout

```
            // Lay out args array onto stack.
            int i = 0;
            while (i < numparams)
            {
                // Push args array reference onto the stack, followed
                // by the current argument index (i). The Ldelem_Ref opcode
                // will resolve them to args[i].

                // Argument 1 of dynamic method is argument array.
                il.Emit(OpCodes.Ldarg_1);
                il.Emit(OpCodes.Ldc_I4, i);
                il.Emit(OpCodes.Ldelem_Ref);

                // If parameter [i] is a value type perform an unboxing.
                Type parmType = parms[i].ParameterType;
                if (parmType.IsValueType)
                {
                    il.Emit(OpCodes.Unbox_Any, parmType);
                }

                i++;
            }
```

We read each element inside the `args` array and push it onto the stack. Unboxing is performed in the case of value type parameters (since the `args` array is a collection of references, we have to convert a reference to a value to the value itself).

Method call

```
            // Perform actual call.
            // If method is not final a callvirt is required
```

```
            // otherwise a normal call will be emitted.
            if (method.IsFinal)
            {
                il.Emit(OpCodes.Call, method);
            }
            else
            {
                il.Emit(OpCodes.Callvirt, method);
            }

            if (method.ReturnType != typeof(void))
            {
                // If result is of value type it needs to be boxed
                if (method.ReturnType.IsValueType)
                {
                    il.Emit(OpCodes.Box, method.ReturnType);
                }
            }
            else
            {
                il.Emit(OpCodes.Ldnull);
            }

            // Emit return opcode.
            il.Emit(OpCodes.Ret);
```

Here we generate the actual method call, process a possible return value, and return from the call site. If our method is final (not virtual nor an interface implementation method) we emit a light `call`, otherwise a `callvirt` is required. If a method return value exists and is a value type, it needs to be boxed, because our `DynamicMethod` delegate returns a generic `object` reference. If the method has a `void` return type, we simply make our delegate return a `null` value, pushing it onto the stack.

Performance Comparisons

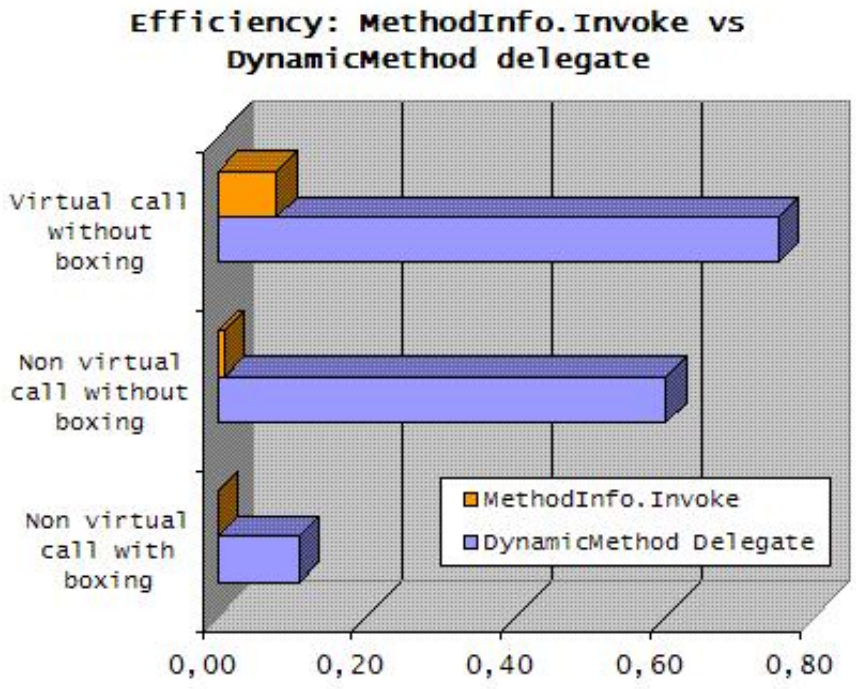
As you can see, the `DynamicMethod` delegate implementation has been kept as simple as possible. The major overhead over a standard early-bound call consists in the need to box the return value when it is a value type. This is a costly operation, because it implies a heap allocation and a copy of the value being boxed. Boxing overhead can occur even during the creation of the input arguments array, but this has not been taken into account in performance measurements, because it is part of the user code.

I measured the efficiency of `DynamicMethod` delegates vs. standard late-bound invocation in three different calling scenarios with the following results (**Note**: 1 is the efficiency of a standard direct call in the same calling scenario):

```
== Testing performance of dynamic method delegates:
== Test call type: static method with boxing on return value
 Results for 5 tests on 500000 iterations:
  Direct method call:       6ms
  Dynamic method delegates: 39ms (Efficiency: 0,15)
  MethodInfo.Invoke:        4616ms (Efficiency: 0,001)

== Testing performance of dynamic method delegates:
== Test call type: static method without boxing on return value
 Results for 5 tests on 100000 iterations:
  Direct method call:       8ms
  Dynamic method delegates: 14ms (Efficiency: 0,57)
  MethodInfo.Invoke:        894ms (Efficiency: 0,009)
```

```
== Testing performance of dynamic method delegates
== Test call type: virtual method without boxing on return value
 Results for 5 tests on 10000 iterations:
  Direct method call:       10ms
  Dynamic method delegates: 13ms (Efficiency: 0,77)
  MethodInfo.Invoke:        166ms (Efficiency: 0,06)
```



`DynamicMethod` delegates' performance proves to be superior (in terms of time efficiency) in all the proposed calling scenarios. When return value boxing occurs, however, both late-bound invocation and hybrid invocation suffer a heavy efficiency loss, as expected.

Conclusion

I've tried to illustrate a simple implementation of one of the ideas exposed inside Joel Polbar's article, along with some observations over the implementation itself, hoping that this work will be at least partly useful in understanding what hybrid invocation is and how it can be achieved. As a conclusive note, I'd like to underline that this technique **is not** a substitute of standard late-bound invocation through Reflection. `DynamicMethod` delegates should be used instead of `MethodInfo.Invoke` or analog solutions only when a method (or property accessor) that isn't known at design time needs to be called with medium-high frequency inside a fully trusted security scenario.

History

- 10-Jul-05 -> First release.

License