Runtime Code Generation with JVM and CLR

Peter Sestoft (sestoft@dina.kvl.dk)

Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Copenhagen, Denmark
and
IT University of Copenhagen

Draft version 1.00 of 2002-10-30

Abstract Modern bytecode execution environments with optimizing just-in-time compilers, such as Sun's Hotspot Java Virtual Machine, IBM's Java Virtual Machine, and Microsoft's Common Language Runtime, provide an infrastructure for generating fast code at runtime. Such *runtime code generation* can be used for efficient implementation of parametrized algorithms. More generally, with runtime code generation one can introduce an additional binding-time without performance loss. This permits improved performance and improved static correctness guarantees.

We report on several experiments with runtime code generation on modern execution platforms. In particular, we show how to introduce C#-style delegates in Java using runtime code generation, to avoid most of the overhead of wrapping and unwrapping method arguments and method results usually incurred by reflective method calls. Furthermore, we give a high-speed implementation of the Advanced Encryption Standard (AES, also known as Rijndael) in C# using runtime code generation. Finally, we experiment with sparse matrix multiplication using runtime code generation on both platforms.

1 Introduction

Runtime code generation has been a research topic for many years, but its practical use has been somewhat limited. Traditionally, runtime code generation tools have been bound not only to a particular language, but also to a particular platform. For instance, runtime code generation for C must take into account the machine architecture, operating system conventions, and compiler-specific issues such as calling conventions. Abstracting away from these details often results in sub-optimal code.

To avoid this platform dependence, runtime code generation has often been done in bytecode execution systems, but then the generated code will not be executed at native speed. This makes it much harder to achieve worthwhile speed-ups.

New bytecode-based execution platforms such as the Java Virtual Machine and Microsoft's Common Language Runtime (CLR) contain just-in-time compilers that generate well-optimized native machine code from the bytecode at runtime. Thus, these platforms combine the advantages of bytecode generation (ease of code generation, portability) with native code generation (speed of the generated code).

Moreover, both platforms incorporate a bytecode verifier that detects type errors and other flaws before executing the bytecode. While the resulting error messages are not particularly detailed ('Operation may corrupt the runtime') they are more useful than a program crash that happens billions of instructions after the runtime was corrupted by flawed code generated at runtime.

1.1 The Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is a specification of a stack-based object-oriented bytecode language. Implementations such as Sun Microsystem's Hotspot Java Virtual Machine[1] [6, 29] and IBM's J2RE Java Virtual Machine[2] [13] execute JVM bytecode by a mixture of interpretation, just-in-time (JIT) generation of native machine code, and adaptive optimizations. There are two versions of the Sun Hotspot JVM, the Client VM, which generates reasonable code fast, and the Server VM, which generates more optimized code more slowly by performing dead code elimination, array bounds check elimination, and so on.

The Java class library does not provide facilities for runtime code generation, but several third-party packages exist, for instance Bytecode Engineering Library BCEL [1] and `gnu.bytecode` [5].

Our experiments suggest that bytecode generated at runtime is usually compiled to machine code that is just as efficient as bytecode compiled from Java source code. So *a priori* there is no performance penalty for generating bytecode at runtime instead of compiletime (from Java source code). For instance, consider a simple loop such as this:

```
do {
  n--;
} while (n != 0);
```

The corresponding JVM bytecode may be:

```
0: iinc 0 -1         // Decrement local variable number 0 (n) by 1
   iload_0           // Load local variable number 0 (n)
   ifne 0            // Go to instruction 0 if variable was non-zero
```

The same loop can be expressed like this, using general bytecode instructions for subtraction, duplication, and comparison:

```
0: iload_0           // Load local variable number 0 (n)
   iconst_1          // Push constant 1
   isub              // Subtract
   dup               // Duplicate result
   istore_0          // Store in local variable number 0 (n)
   iconst_0          // Push constant 0
   if_icmpgt 0       // Go to instruction 0 if n > 0
```

Although the latter code sequence is twice as long, the just-in-time compiler generates equally fast machine code, and this code is just as fast as code compiled from the Java loop shown above.

Sun's Hotspot Client VM performs approximately 240 million iterations per second, and the IBM JVM performs more than 400 million iterations per second. In both cases this is just as fast as bytecode compiled from Java source, and in case of the IBM JVM, as fast as optimized code compiled from C; see Figure 1. This shows that bytecode generated at runtime carries no inherent speed penalty compared to code compiled from Java programs. The Sun Hotspot Server VM optimizes more aggressively, and removes the entire loop because it is dead code.

On the same platform, straightline JVM bytecode can be generated at a rate of about 200,000 bytecode instructions per second with Sun HotSpot Client VM and 180,000 bytecode instructions per second with the IBM JVM, using the `gnu.bytecode` package [5]; the BCEL package [1] is only half as fast. The code generation time includes the time taken by the just-in-time compiler to generate machine code.

[1] We use Sun HotSpot Client VM and Server VM 1.4.0 under Linux on an 850 MHz Mobile Pentium 3.

[2] We use IBM J2RE JVM 1.3.1 under Linux on an 850 MHz Mobile Pentium 3.

	Sun HotSpot		IBM	MS	C
	Client	Server	JVM	CLR	gcc -O2
Compiled loop (million iter/sec)	243	∞	421	408	422
Generated loop (million iter/sec)	243	∞	421	408	N/A
Code generation (thousand instr/sec)	200	142	180	100	N/A

Figure 1: Code speed for simple loop, and code generation speed.

In the Java Virtual Machine, the size of a method body is limited to 65535 bytes of bytecode. It is usually pointless to generate methods larger than that at runtime, but this limitation sometimes complicates experiments.

The Sun HotSpot JVM and the IBM JVM are available for free for the operating systems Linux, Microsoft Windows, Solaris, HP-UX, Apple MacOS X, IBM AIX, and IBM OS/390, and for a range of processor types.

Other implementations of the JVM with just-in-time compilation are available, but we have not tested their suitability for runtime code generation.

Some notable JVM implementations suitable for experimentation are IBM's Jikes Research Virtual Machine (RVM, formerly Jalapeño) in which a range of dynamic optimizations may be turned on or off [14], and Intel's Open Platform Runtime [4].

1.2 The Microsoft Common Language Runtime (CLR)

Another major bytecode execution platform is Microsoft's Common Language Runtime (CLR), part of the so-called .Net platform. Like the Java Virtual Machine it uses a stack-based object-oriented bytecode, and natively supports runtime code generation using the .Net Framework classes in namespace `System.Reflect.Emit`.

Experiments confirm that CLR bytecode generated at runtime is no less efficient than bytecode compiled from C#. Under Microsoft's CLR[3], bytecode generated at runtime performs more than 400 million iterations per second in the simple `do-while` loop shown in the preceding section, which is just as fast as code compiled from C#. The Microsoft CLR just-in-time compiler appears to generate code that is comparable in speed to that of the IBM JVM and somewhat faster than Sun HotSpot Client VM; see Figure 1. The experiment reported in Section 4 shows that this expectation scales to more realistic programs also. Probably we underestimate the real of MS CLR performance slightly, because we run MS Windows under VmWare, not on the bare machine.

The Microsoft CLR seems more sensitive than the JVMs to the actual choice of bytecode instructions. For instance, loop bytecode that involves `dup` operations on the evaluation stack performs only 250 million iterations per second, 37 percent less than the first bytecode sequence shown in the preceding section. Multiple load instructions are clearly preferable to a load followed by a `dup`. Moreover, the speed shows large but reproducible variations depending on what other code can actually be executed (later) by the program.

Straightline CLR bytecode can be generated at a rate of about 100,000 bytecode instructions per second, including the time used by the just-in-time compiler to generate x86 instructions from the bytecode. Most of the code generation time (around 95 per cent) is spent in the just-in-time compiler, but the exact proportion probably depends on the composition of the code.

So far, CLR runtime code generation facilities are available only from Microsoft and for MS Windows and FreeBSD, but as the Mono project [3] develops, they will become available for a range of platforms.

[3]We use Microsoft CLR version 1.0 SP2 under Windows 2000 under VmWare 3.0 under Linux with an 850 MHz Mobile Pentium 3.

1.3 Related work

Runtime code generation has been used for many years in the Lisp, Scheme, and Smalltalk communities. The backquote and comma notation of Lisp and Scheme originates from MIT Lisp Machine Lisp (1978), and may be inspired by W.v.O. Quine's quasiquotation (1960). Backquote and comma are classical tools for code generation, which together with an `eval` or `compile` function provides a means for runtime code generation.

A number of technical reports by Keppel, Eggers and Henry argued the utility of runtime code generation also in other programming languages [25, 26]. Later work in the same group led to the DyC system for runtime code generation in C [8, 22].

Engler and others [18] developed very fast tools for runtime code generation in C. An untyped two-level language 'C (tick-C) was based on the Vcode library [19]. Neither Vcode nor 'C is maintained any longer (2002). The Gnu Lightning library [2] may be considered a replacement for Vcode; it currently supports portable runtime code generation in C for the Intel x86, Sun Sparc, and PowerPC processor architectures.

Leone and Lee developed Fabius, a system for runtime code generation in Standard ML, implemented in the SML/NJ system [27]. It is no longer maintained.

The Tempo specializer from Consel's group is a partial evaluator for C that can generate specialized programs at runtime, generating machine code using so-called code templates [12].

Bytecode generation tools for Java include `gnu.bytecode` [5], developed for the Kawa implementation of Scheme, and the Bytecode Engineering Library (BCEL) [1], formerly called JavaClass, which is used in several projects.

Oiwa, Masuhara and Yonezawa [31] present a Java-based typed two-level language for expressing dynamic code generation in Java, rather than using JVM bytecode. The code generation speed and the runtime results reported in that paper are however somewhat disappointing.

The Microsoft .Net Framework Library uses runtime code generation, for instance in the implementation of regular expressions in namespace `System.Text.RegularExpressions`. The source code of this implementation is available as part of the shared source implementation of the CLR (also known as Rotor).

Cisternino and Kennedy developed a library for C# that considerably simplifies runtime generation of CIL code [11]. It uses C# custom attributes and requires a modification of the CLR.

At least two libraries for runtime bytecode generation in Caml bytecode exist, namely Rhiger's bytecode combinators [32], and Lomov and Moskal's Dynamic Caml [28]. Dynamic Caml permits runtime code generating programs to be written using a two-level source language syntax, so one does not have to work at the bytecode level.

MetaML [33] and MetaOCaml [10] are typed multi-level languages based on Standard ML and OCaml.

The first type system for a multi-stage language was the two-level lambda calculus proposed by Nielson [30]. Davies and Pfenning generalized Nielson's type system, relating it to the modal logic S4. An extension of this type system was used by Wickline, Lee and Pfenning in a typed multi-stage version of ML [34].

Runtime code generation is related to generative programming [15], staged computation [24] and partial evaluation [23]. Just-in-time compilation and dynamic compilation (as in the Sun HotSpot JVMs and Microsoft's CLR) are themselves instances of runtime code generation. Machine code is generated just before it is to be executed, usually from a higher-level bytecode language, and possibly taking into account information available only at runtime, such as method call patterns, the actual CPU type (AMD Athlon or Intel Pentium 2, 3 or 4), and so on.

The implementation of delegates for efficient reflective method calls in Java shown in Section 3 is related to Breuel's work on implementing dynamic language features in Java [9]. That paper hints at the possibility of creating delegates in analogy to other constructions in the paper. However, the Dynamic toolkit described in Breuel's paper does not appear to be available.

2 Examples of runtime code generation

This section shows a few examples of runtime code generation for Microsoft's CLR using C#.

2.1 Evaluation of polynomials

Consider the evaluation of a polynomial $p(x)$ for many different values of x. The polynomial $p(x)$ of degree n in one variable x and with coefficent array $cs[\,]$ is defined as follows:

$$p(x) = cs[0] + cs[1] \cdot x + cs[2] \cdot x^2 + \cdots + cs[n] \cdot x^n$$

According to Horner's rule, this formula is equivalent to:

$$p(x) = cs[0] + x \cdot (cs[1] + x \cdot (\ldots (x \cdot (cs[n] + x \cdot 0)) \ldots))$$

Therefore $p(x)$ can be computed by this `for`-loop, which evaluates the above expression inside-out and stores the result in variable `res`,

```
double res = 0.0;
for (int i=cs.Length-1; i>=0; i--)
  res = res * x + cs[i];
return res;
```

If we need to evaluate a polynomial with fixed coefficents $cs[\,]$ for a large number of different values of x, then it may be worthwhile to unroll this loop to a sequence of assignments, where `cs_i` is the contents of array cell $cs[i]$:

```
double res = 0.0;
res = res * x + cs_n;
...
res = res * x + cs_1;
res = res * x + cs_0;
return res;
```

These source statements could be implemented by stack-oriented bytecode such as this:

```
Ldc_R8 0.0                          // push res = 0.0 on stack
Ldarg_1                             // load x
Mul                                 // compute res * x
Ldc_R8 cs_n                         // load cs[n]
Add                                 // compute res * x + cs[n]
...
Ldarg_1                             // load x
Mul                                 // compute res * x
Ldc_R8 cs_0                         // load cs[0]
Add                                 // compute res * x + cs[0]
Return                              // return res
```

This bytecode can be generated at runtime using classes from the namespace `System.Reflection.Emit` in the CLR Framework, as follows:

```
ilg.Emit(OpCodes.Ldc_R8, 0.0);              // push res = 0.0 on stack
for (int i=cs.Length-1; i>=0; i--) {
  ilg.Emit(OpCodes.Ldarg_1);                // load x
  ilg.Emit(OpCodes.Mul);                    // compute res * x
  ilg.Emit(OpCodes.Ldc_R8, cs[i]);          // load cs[i]
  ilg.Emit(OpCodes.Add);                    // compute res * x + cs[i]
}
ilg.Emit(OpCodes.Ret);                      // return res;
```

The `ilg` variable holds a bytecode (Intermediate Language) generator, representing a method body to be generated. Executing `ilg.Emit(...)` appends a bytecode instruction to the instruction stream. The `Ldc_R8` instruction loads a floating-point constant onto the evaluation stack. The `Ldarg_1` instruction loads method parameter number 1, assumed to hold *x*, onto the stack. The `Mul` instruction multiplies the two top-most stack elements, and the `Add` instruction adds the two top-most stack elements, leaving the result on the stack top.

The generated code will be a linear sequence of instructions for pushing a constant or variable, or for multiplying or adding stack elements; it contains no loops, tests, or array accesses. All loop tests and array accesses are performed at code generation time.

As a further optimization, when a coefficient $cs[i]$ is zero, pushing and adding it has no effect and no code needs to be generated for it. The code generator is easily modified to perform this optimization:

```
ilg.Emit(OpCodes.Ldc_R8, 0.0);              // push res = 0.0 on stack
for (int i=cs.Length-1; i>=0; i--) {
  ilg.Emit(OpCodes.Ldarg_1);                // load x
  ilg.Emit(OpCodes.Mul);                    // compute res * x
  if (cs[i] != 0.0) {
    ilg.Emit(OpCodes.Ldc_R8, cs[i]);        // load cs[i]
    ilg.Emit(OpCodes.Add);                  // compute x * res + cs[i]
  }
}
ilg.Emit(OpCodes.Ret);                      // return res;
```

Runtime code generation is interesting in the polynomial example because there are two binding-times in the input data. The coefficient array $cs[]$ is available early, whereas the value of the variable *x* is available only later.

Another way of saying this is that $cs[]$ remains fixed over a large number of different values of *x*, so specialization of the general loop code with respect to $cs[]$ is worthwhile. This permits *staging*: In stage one, the coefficient array *cs* is available, and all computations depending only that array are performed. In stage two, the computations depending also on *x* are performed.

Staging by runtime code generation gives a speed-up over the straightforward implementation already for polynomials of degree *n* higher than 4, and especially if some coefficients $cs[i]$ are zero. Speed-up factors of 2 to 4 appear to be typical.

2.2 The power function

For another example, consider computing x^n, that is, x raised to the n'th power, for integers $n \geq 0$ and arbitrary x. This example is a classic in the partial evaluation literature. The C# or Java function `Power(n, x)` below computes x^n:

```
public static int Power(int n, int x) {
  int p;
  p = 1;
  while (n > 0) {
    if (n % 2 == 0)
      { x = x * x; n = n / 2; }
    else
      { p = p * x; n = n - 1; }
  }
  return p;
}
```

The function relies on these equivalences, for n even ($n = 2m$) and n odd ($n = 2m+1$):

$$\begin{aligned} x^{2m} &= (x^2)^m \\ x^{2m+1} &= x^{2m} \cdot x \end{aligned}$$

Note that the `while`- and `if`-conditions in `Power` depend only on n, so for a given value of n one can unroll the `while`-loop and eliminate the `if`-statement. Thus if we have a fixed value of n and need to compute x^n for many different values of x, we can generate a specialized function `power_n(x)` that takes only one parameter x and avoids all the tests and computations on n.

The following C# method `PowerGen` takes n as argument and generates the body of a `power_n` instance method, using a bytecode generator `ilg` for the method being generated. The `PowerGen` method performs the computations that depend only on n, and generates code that will later perform the computations involving x and p:

```
public static void PowerGen(ILGenerator ilg, int n) {
  ilg.DeclareLocal(typeof(int));      // declare p as local_0
  ilg.Emit(OpCodes.Ldc_I4_1);
  ilg.Emit(OpCodes.Stloc_0);          // p = 1;
  while (n > 0) {
    if (n % 2 == 0) {
      ilg.Emit(OpCodes.Ldarg_1);      // x is arg_1
      ilg.Emit(OpCodes.Ldarg_1);
      ilg.Emit(OpCodes.Mul);
      ilg.Emit(OpCodes.Starg_S, 1);   // x = x * x
      n = n / 2;
    } else {
      ilg.Emit(OpCodes.Ldloc_0);
      ilg.Emit(OpCodes.Ldarg_1);
      ilg.Emit(OpCodes.Mul);
      ilg.Emit(OpCodes.Stloc_0);      // p = p * x;
      n = n - 1;
    }
  }
  ilg.Emit(OpCodes.Ldloc_0);
  ilg.Emit(OpCodes.Ret);              // return p;
}
```

Note that the structure of `PowerGen` is very similar to that of `Power`. The main difference is that operations that depend on the *late* or *dynamic* argument *x* have been replaced by actions that generate bytecode, whereas operations that depend only on the *early* or *static* argument *n* are executed as before. Roughly, `PowerGen` could be obtained from `Power` just by keeping all code that can be executed using only early or static information (variable *n*), and replacing the remaining code by bytecode-generating instructions.

A method such as `PowerGen` is called the *generating extension* of `Power`. Given a value for `Power`'s static argument *n*, it generates a version of `Power` specialized for that value.

For $n = 16$, the specialized method generated by `PowerGen` is considerably faster than the general method `Power`; see Figure 2. The fastest way to call the generated specialized method is to use an interface call; a reflective call is very slow, on the other hand. The same pattern is seen in Java. Delegate calls (in C#) are twice as slow as interface calls, apparently.

In a Java implementation of this example, calling the specialized method can be 4 times faster than calling the general method when using the Sun HotSpot Client VM, approximately 8 times faster when using the Sun HotSpot Server VM, and fully 22 times faster with the IBM JVM. This very high speed-up factor is probably achieved by inlining the specialized method code during machine code generation, so similar speed-ups cannot be expected when larger or more complex code is generated.

	Sun HotSpot		**IBM**	**MS**
	Client	**Server**	**JVM**	**CLR**
Reflective call to specialized ($n = 16$)	8.225	2.041	24.235	26.879
Interface call to specialized ($n = 16$)	1.334	0.094	0.026	0.260
Delegate call to specialized ($n = 16$)	N/A	N/A	N/A	0.541
Static call to general method	5.369	0.759	0.711	0.711
Interface call to general method	5.677	0.830	0.571	0.741
Delegate call to general method	N/A	N/A	N/A	1.062

Figure 2: Time in seconds for 10 million calls to specialized and general power methods.

2.3 Practical CLR bytecode generation

In the examples above we have focused on generating the bytecode for a method body. Here we see how to set up the context for that bytecode, and how to execute it. The goal is to generate some method `MyMethod` in some class `MyClass`, as if declared by

```
class MyClass : IMyInterface {
  public MyClass() : base() { }

  public double MyMethod(double x) {
    ... method body generated using ilg.Emit(...) ...
  }
  ...
}
```

The method's body consists of the bytecode generated using `ilg`. To be able to call the generated method `MyMethod` efficiently, `MyClass` must implement an interface `IMyInterface` that describes the method. The idea is that we can create an instance of the class, cast it to the interface, and then call the method `MyMethod` on that instance. For this to work, the interface must describe the generated method:

```
interface IMyInterface {
  double MyMethod(double x);
}
```

Moreover, the class must have a constructor `MyClass()` as shown above, so that one can create instances of the class. The constructor's only action is to call the superclass constructor.

In the CLR, a class must belong to some module, and a module must belong to an assembly. Hence before one can generate any code, one needs an `AssemblyBuilder`, a `ModuleBuilder`, a `TypeBuilder` (for the class), and a `MethodBuilder`, as outlined below. The `ILGenerator ilg` is then obtained from the `MethodBuilder`, with the purpose of generating the method's body:

```
AssemblyName assemblyName = new AssemblyName();
AssemblyBuilder assemblyBuilder = ... assemblyName ...
ModuleBuilder moduleBuilder = assemblyBuilder.DefineDynamicModule(...);
TypeBuilder typeBuilder = moduleBuilder.DefineType("MyClass", ...);
... (1) generate a constructor in class MyClass ...
MethodBuilder methodBuilder = typeBuilder.DefineMethod("MyMethod", ...);
ILGenerator ilg = methodBuilder.GetILGenerator();
... (2) use ilg to generate the body of method MyClass.MyMethod ...
```

The class must have a constructor so that we can make an instance of it. An argumentless constructor that simply calls the superclass (`Object`) constructor can be built using this boilerplate code:

```
ConstructorBuilder constructorBuilder =
  typeBuilder.DefineConstructor(MethodAttributes.Public,
                                CallingConventions.HasThis,
                                new Type[] { });
ILGenerator ilg = constructorBuilder.GetILGenerator();
ilg.Emit(OpCodes.Ldarg_0);      // push the current object, 'this'
ilg.Emit(OpCodes.Call, typeof(Object).GetConstructor(new Type[] {}));
ilg.Emit(OpCodes.Ret);
```

After the method body has been generated, the class must be created by a call to method `CreateType`:

```
Type ty = typeBuilder.CreateType();
```

The object bound to `ty` represents the newly built class `MyClass`, containing the instance method `MyMethod`. To call the method, create an instance `obj` of the class, cast it to the interface `IMyInterface`, and call the method in that object. To create an instance of the class we obtain the class's argumentless constructor and call it using the reflection facilities of CLR:

```
Object obj = ty.GetConstructor(new Type[] {}).Invoke(new Object[] { });
IMyInterface myMethod = (IMyInterface)obj;
double res = myMethod.MyMethod(3.14);
```

The interface method call is fast because no argument wrapping or result unwrapping is needed: the argument 3.14 is passed straight to the generated bytecode.

Alternatively, one can use the CLR reflection facilities to get a handle `m` to the generated method by evaluating the expression `ty.GetMethod("MyMethod")`. Using the handle, one can then call the method, passing arguments in an object array, and getting the results as an object:

```
MethodInfo m = ty.GetMethod("MyMethod");
double res = (double)m.Invoke(null, new object[] { 3.14 });
```

However, reflective calls to a method handle are inefficient because of the need to wrap the method's arguments as an object array, and similarly unwrap the method result. This wrapping and unwrapping is syntactically implicit in C#, but takes time and space even so.

A third way to call the generated method is to turn it into a so-called delegate (a typed function reference); this avoids the argument wrapping and result unwrapping but still turns out to be slower than the interface method call. A delegate such as `myMethod` can be called directly without any wrapping or unwrapping of values:

```
D2D myMethod = (D2D)Delegate.CreateDelegate(typeof(D2D),
                                            obj,
                                            ty.GetMethod("MyMethod"));
double res = myMethod(3.14);
```

The example above assumes that the delegate type `D2D` describes a function that takes a `double` argument *x* and returns a `double` result. The delegate type `D2D` may be declared as follows in C#:

```
public delegate double D2D(double x);
```

Other delegate (function) types may be described similarly.

2.4 Practical Java bytecode generation with `gnu.bytecode`

Here we show the necessary setup for generating a Java class `MyClass` containing a method `MyMethod`, using the `gnu.bytecode` package [5]. An object `co` representing a named class must be created, with specified superclass and access modifiers. As in Section 2.3 it must also be declared to implement an interface describing the generated method:

```
ClassType co = new ClassType("MyClass");
co.setSuper("java.lang.Object");
co.setModifiers(Access.PUBLIC);
co.setInterfaces(new ClassType[] { new ClassType("IMyInterface") });
... (1) generate a constructor in class MyClass ...
Method mo = co.addMethod("MyMethod");
mo.setSignature("(D)D");
mo.setModifiers(Access.PUBLIC);
mo.initCode();
CodeAttr jvmg = mo.getCode();
... (2) use jvmg to generate the body of method MyClass.MyMethod ...
```

An argumentless constructor is added to class `MyClass` (step 1 above) by adding a method with the special name `<init>`. The constructor should just call the superclass constructor, using an `invokespecial` instruction:

```
Method initMethod =
  co.addMethod("<init>", new Type[] {}, Type.void_type, 0);
initMethod.setModifiers(Access.PUBLIC);
initMethod.initCode();
CodeAttr jvmg = initMethod.getCode();
Scope scope = initMethod.pushScope();
Variable thisVar = scope.addVariable(jvmg, co, "this");
jvmg.emitLoad(thisVar);
jvmg.emitInvokeSpecial(ClassType.make("java.lang.Object")
                                  .getMethod("<init>", new Type[] {}));
initMethod.popScope();
jvmg.emitReturn();
```

Then a method represented by method object `mo` must be added to the class (step 2 above), with given signature and access modifiers, and a code generator `jvmg` is obtained for the method and is used to generate the method body. An example use of code generators in `gnu.bytecode` can be seen in Section 5.2.

Once the constructor and the method have been generated, a representation of the class is written to a byte array and loaded into the JVM using a class loader. This produces a class reference `ty` representing the new class `MyClass`:

```
byte[] classFile = co.writeToArray();
Class ty = new ArrayClassLoader().loadClass("MyClass", classFile);
```

An instance `obj` of the class is created and cast to the interface describing the generated method, and then the method in the object is called using an interface call:

```
Object obj = ty.newInstance();
IMyInterface myMethod = (IMyInterface)obj;
double res = IMyInterface.MyMethod(3.14);
```

Alternatively, one can use reflection on `ty` to obtain an object `m` representing the method in the class, and then call that method:

```
java.lang.reflect.Method m =
  ty.getMethod("MyMethod", new Class[] { double.class });
Double ro = (Double)m.invoke(obj, new Object[] { new Double(3.14) });
double res = ro.doubleValue();
```

As can be seen, this requires wrapping of arguments and unwrapping of results, which is costly. Recall that in Java, `double.class` is an object that represents the primitive type `double` at runtime. As in C#, reflective method calls are slow because of the wrapping and unwrapping, but in Java there is no built-in way to avoid them.

2.5 Practical Java bytecode generation with BCEL

The Bytecode Engineering Library BCEL [1] is another third-party Java library that can be used for runtime code generation. Here we outline the necessary setup for code generation with BCEL.

One must create a class generator `cg`, specifying superclass, access modifiers and a constant pool `cp` for the generated class, and an interface describing the generated method. Then one must generate a constructor for the class, and the method:

```
ClassGen cg = new ClassGen("MyClass", "java.lang.Object",
                           "<generated>",
                           Constants.ACC_PUBLIC | Constants.ACC_SUPER,
                           new String[] { "Int2Int" });
InstructionFactory factory = new InstructionFactory(cg);
ConstantPoolGen cp = cg.getConstantPool();
... (1) generate a constructor in class MyClass ...
InstructionList il = new InstructionList();
MethodGen mg = new MethodGen(Constants.ACC_PUBLIC | Constants.ACC_STATIC,
                             Type.DOUBLE, new Type[] { Type.DOUBLE },
                             new String[] { "x" },
                             "MyMethod",
                             "MyClass",
                             il, cp);
... (2) use il to generate the body of method MyClass.MyMethod ...
mg.setMaxStack();
cg.addMethod(mg.getMethod());
```

An argumentless constructor is added to class `MyClass` (step 1 above) by adding a method with the special name `<init>`. The constructor should just call the superclass constructor:

```
InstructionFactory factory = new InstructionFactory(cg);
InstructionList ilc = new InstructionList();
MethodGen mgc = new MethodGen(Constants.ACC_PUBLIC,
                              Type.VOID,
                              new Type[] { }, new String[] { },
                              "<init>", "MyClass",
                              ilc, cp);
ilc.append(factory.createLoad(Type.OBJECT, 0));
ilc.append(factory.createInvoke("java.lang.Object", "<init>",
                                Type.VOID,
                                new Type[] { },
                                Constants.INVOKESPECIAL));
ilc.append(new RETURN());
mgc.setMaxStack();
cg.addMethod(mgc.getMethod());
```

When a method body has been generated (step 2 above), we can write a representation `clazz` of the class to a byte array and then load it into the JVM using a class loader. This gives a class reference `ty` representing the new class `MyClass`:

```
JavaClass clazz = cg.getJavaClass();
byte[] classFile = clazz.getBytes();
Class ty = new ArrayClassLoader().loadClass("MyClass", classFile);
```

As in Section 2.4, an instance of the class is created and cast to the interface describing the generated method, and then the method in the object is called using an interface call:

```
Object obj = ty.newInstance();
IMyInterface myMethod = (IMyInterface)obj;
double res = IMyInterface.MyMethod(3.14);
```

3 Efficient reflective method calls in Java

This section shows that runtime code generation can be used to introduce the concept of a *delegate* (typed function reference) in the Java programming language. Delegates are known from C# and other programming languages. The experiments below show that delegates can be used to improve the speed of Java reflective method calls by a factor of 8 to 16.

This is primarily of interest when a pre-existing method needs to be called by reflection several thousand times. A method generated at runtime should be called by interface calls through an interface implemented by the generated class to which the method belongs, as shown in Sections 2.3 through 2.5. This is far more efficient than a reflective call, and also more efficient than a delegate call as implemented here.

3.1 Reflective method calls are slow

In Java one can obtain an object `mo` of class `java.lang.reflect.Method` representing a named method from a named class, and one can call that method using a reflective method call `mo.invoke(...)`. However, the arguments to method `mo` must be given as an array of objects, and the result is returned as an object; this means that primitive type arguments must be wrapped as objects, and primitive type results must be unwrapped.

This wrapping of arguments and unwrapping of results imposes a considerable overhead on reflective method calls. A reflective call to a public method in a public class appears to be 16 to 28 times slower than a direct call. See Figure 3. The slowdown is higher for methods taking arguments than for those that do not, because of the argument wrapping and unwrapping. The reflective slowdown is approximately the same regardless of whether the called method is static, virtual, or called via an interface. A reflective call to a public method in a non-public class is slower by a further factor of 8, probably due to access checks on the class.

	Static method		**Instance method**		
	Reflective	**Direct**	**Reflective**	**Virtual**	**Interface**
No arguments or results	3.279	0.288	3.39	0.329	0.372
Integer argument and result	7.746	0.310	8.18	0.346	0.343

Figure 3: Time in seconds for 10 million method calls, using Sun HotSpot Client VM.

The figure shows that reflective method calls carry a considerable performance overhead. In Microsoft's CLR, most of this overhead can be avoided by turning the reflective method reference into a delegate (Section 2.3). Java does not have a built-in notion of delegate, but the next sections show how to add delegates to the Java programming language using runtime code generation. This way, most of the call overhead can be avoided.

3.2 Delegate objects in Java

To implement delegates, we need an interface describing the delegate by a method `invoke` with appropriate argument and result types. We generate a new class that implements that interface and whose `invoke` method calls the method obtained by reflection, and then we create an instance `dlg` of the generated class and cast it to the interface. A call to `dlg.invoke` will call the method obtained by reflection, yet without the overhead of wrapping and unwrapping.

Consider a method `m` from a class `C` with return type `R` and parameter types `T1`, ..., `Tn`. Assume for now that the method is static; our technique works also for instance methods (Section 3.4):

```
public class C {
  public static R m(T1 x1, ..., Tn xn) { ... }
  ...
}
```

Using Java's reflection facilities one can obtain a method object `mo` of class `java.lang.reflect.Method`, and then one can call method `m` by invoking `mo.invoke(wrargs)`, where `wrargs` is an array of objects holding the argument values `v1`, ..., `vn`. Thus any arguments of primitive type (`int`, `double`, and so on) must be wrapped as objects. The call will return the method's result (if any) as an object, which must be cast to `m`'s result type `R`, or unwrapped in case `R` is a primitive type.

Instead of calling `m` via the method reference `mo`, with wrapping and unwrapping of arguments, we shall create and use a delegate object `dlg`. The delegate object will be made to implement a user-declared interface `OI` that describes a method `invoke` with precisely the parameter types and result types of `m`, so `dlg.invoke` and `m` have the same signature:

```
interface OI {
  R invoke(T1 x1, ..., Tn xn);
}
```

Then one can call `m(v1, ..., vn)` as `dlg.invoke(v1, ..., vn)` via the delegate `dlg`, without any wrapping of the argument values `v1`, ..., `vn`, nor unwrapping of results.

The implementation of this idea is detailed in the sections below. The efficiency of the approach is illustrated in Figure 4, which should be read as additional columns for Figure 3 above. We see that a delegate call to a method without arguments is 8 times faster than an ordinary reflective call. The speed-up factor is closer to 16 for a method with arguments, because the reflective call's argument wrapping is avoided in the delegate call. A call via a delegate is still around 50 percent slower than a direct call, but this slowdown is quite similar to that incurred by delegate calls in CLR.

	Static method Delegate	**Instance method Delegate**
No arguments or results	0.446	0.477
Integer argument and result	0.487	0.537

Figure 4: Time in seconds for 10 million delegate calls, using Sun HotSpot Client VM.

The speed-up derives from two sources: (1) access permissions and types are checked once and for all when the delegate is created, not at every call; and (2) the object allocation required by argument wrapping and the downcasts implied by result unwrapping are avoided completely. With the IBM JVM, reflective calls are 3 times slower than with Sun HotSpot Client VM, but delegate calls are 10 times slower. Thus with the IBM JVM, delegate calls are only 3 to 5 times faster than reflective calls.

Note the importance of the user-declared interface `OI`. It enables the compiler to statically typecheck every call `dlg.invoke(...)` to the delegate, although the delegate object `dlg`, and the class of which it is an instance, will be created only at runtime. Our delegate implementation makes sure (and the bytecode verifier checks) that the delegate actually implements the interface `OI`.

3.3 The implementation of delegate creation

Our implementation of delegate objects in Java uses the `gnu.bytecode` [5] package, but could use the BCEL [1] package instead. Assume as above that `mo` is a `java.lang.reflect.Method` object referring to a static

method `m` in a class `C`:

```
public class C {
  public static R m(T1 x1, ..., Tn xn) { ... }
  ...
}
```

Assume further that there is an interface `OI` describing a method `invoke` with the same (unwrapped) return type and parameter types as `m`:

```
interface OI {
  R invoke(T1 x1, ..., Tn xn);
}
```

We create, at runtime, a new class `Dlg` that implements `OI`, and whose `invoke` method simply calls the real method `C.m` represented by `mo` and returns its result, if any. In Java syntax, the class `Dlg` would look like this (assuming that `m`'s return type `R` is non-`void`):

```
public class Dlg implements OI extends Object {
  public Dlg() { super(); }

  public R invoke(T1 p1, ..., Tn pn) {
    return C.m(p1, ..., pn);
  }
}
```

The new class `Dlg` (which is generated by the bytecode tools, not in Java source form) is loaded into the Java Virtual Machine, and an instance `dlg` of class `Dlg` is created; this is the delegate. By construction, `dlg` can be cast to the interface `OI`, and calling `dlg.invoke(...)` has the same effect as calling `mo.invoke(...)`, but avoids the overhead of wrapping and result unwrapping arguments and result. The only overhead is an additional interface method call.

We implement delegate creation by a static method `createDelegate`. A delegate `dlg` that corresponds to method object `mo` and implements interface `OI` can be created and called as follows:

```
OI dlg = (OI)(Delegate.createDelegate(OI.class, mo));
... dlg.invoke(v1, ..., vn) ...
```

Note the cast to interface `OI`, and recall that in Java, `OI.class` is an object of class `java.lang.Class` representing the interface type `OI` at runtime.

More precisely, our implementation of `createDelegate(iface, mo)` performs the following steps:

1. Check that `iface` represents a public interface that has a single method `invoke`.
2. Check that the `invoke` method in the interface has the same parameter and result types as the method represented by `mo`.
3. Check that `mo` represents a public static method `m` (or a public instance method, see Section 3.4) in a public classs `C`.
4. Create a `gnu.bytecode.ClassType` object representing a new public class `Dlg` that implements the interface represented by `iface`.

5. Generate a public argumentless constructor `Dlg()` and add it to the class.

6. Generate a public method `invoke` and add it to the class. The method has the same parameter and result types as `invoke` in the interface `iface`.

7. Generate JVM bytecode for `invoke`'s body. The code contains instructions to call method `C.m` with precisely the arguments passed to `invoke`, and ends with a return instruction of the appropriate kind.

8. Write a representation of the new class `Dlg` to a byte array.

9. Invoke a classloader to load the `Dlg` class from the byte array into the JVM. The JVM bytecode verifier will check the new class before loading it.

10. Use reflection to create an object `dlg` of class `Dlg`, and return that object as the result of the call to `createDelegate(iface, mo)`.

The checks make sure that the generated bytecode passes the bytecode verifier.

Experiments indicate that the time to create a `Dlg` class and a delegate using this approach is approximately 1.5 ms (with the Sun HotSpot Client VM). Thus a delegate must be called approximately 2000 times before the cost of creating the delegate has been recovered. Not surprisingly, this prototype implementation of delegate creation in Java is approximately 100 times slower than the built-in delegate creation in Microsoft's CLR.

Creating a `Dlg` class and a delegate as above consumes approximately 1100 bytes of memory in the Sun HotSpot Client VM. However, a separate `Dlg` class needs to be created only for each distinct method `C.m`. Hence the `createDelegate` method should cache `Dlg` classes (using a `HashMap`, for instance) and reuse them when creating new delegates for a method already seen. This is especially important when creating delegates over instance methods (see below), where many delegates may be created from the same method.

3.4 Implementing delegate objects for instance methods

The previous section described the implementation of delegates for static methods. The procedure to create a delegate for an instance method `m` is almost the same, only the creation of the delegate must store a receiver object in the delegate object `dlg`.

Assume we have a method object `mo` representing an instance method `m` of class `C`, with return type `R` and parameter types `T1`, ..., `Tn`:

```
class C {
  public R m(T1 x1, ..., Tn xn) { ... }
  ...
}
```

Also assume that the (receiver) object `obj` is an object of class `C` or one of its subclasses. The programmer must declare an interface `OI` describing a method `invoke` with the same argument types and result type as `m`, precisely as in Section 3.3.

Now the call `createDelegate(OI.class, mo, obj)` should return an object `dlg` that can be cast to interface `OI`, and so that a call to `dlg.invoke(...)` will call the method as if by `obj.mo(...)`.

This works as above, except that the class `Dlg` constructed at runtime must contain a reference to the receiver object `obj` of class `C`, and must use it when invoking `m`, so the class `Dlg` would look like this in Java syntax:

```
public class Dlg implements I extends Object {
  private C obj;

  public Dlg() { super(); }

  public void setObj(Object obj) {
    this.obj = (C)obj;
  }

  public R invoke(T1 p1, ..., Tn pn) {
    return obj.m(p1, ..., pn);
  }
}
```

The new class `Dlg` is loaded into the JVM using a class loader, an instance `dlg` of the class is created, and the `dlg.setObj(obj)` method is called on the given receiver object `obj`. Now `dlg` is the delegate for `obj.m(...)` and is returned as the result of the call to `Delegate.createDelegate(...)`.

Instead of using a `setObj` method, one might create a one-argument constructor in class `Dlg` and pass `obj` when creating the delegate object. This would complicate the class generator slightly.

4 Efficient implementation of the Advanced Encryption Standard

The Advanced Encryption Standard (AES) [7], also known as the Rijndael algorithm after its inventors J. Daemen and V. Rijmen, is the US Federal standard for encryption of sensitive (unclassified) information. It succeeds the DES encryption algorithm and is expected to be adopted also by the private sector, and internationally.

Starting from an efficient baseline implementation of AES in C#, we have made an implementation that uses runtime code generation in CLR to create a specialized block encryption/decryption routine for a given key. The same bytecode generator generates both the encryption routine and the decryption routine. It works for all three standard key sizes (128, 192, 265 bit), and a block size of 128 bit.

The specialized code generated at runtime (for a known key) is 35 percent faster than the best we could write by hand (for a fixed key size, but unknown key). The generated code can encrypt or decrypt approximately 139 Mbit/s under the Microsoft CLR. Extrapolating from data given by Rijmen and Daemen, a highly optimized implementation by Brian Gladman using Visual C++ and native Pentium 3 rotate instructions (not expressible in ANSI C nor in C#) can encrypt 280–300 Mbit/s [16, 21]. Given that our implementation runs in a managed execution environment, its performance is wholly satisfactory.

4.1 Brief description of the AES

The AES algorithm is a block cipher, that is, it encrypts data in blocks of 128 bits (16 bytes) at a time. The algorithm works in two phases:

(1) Given an encryption or decryption key (of size 128, 192 or 256 bits), create an array `rk[]` of so-called round keys. Each round key can be considered a 4 by 4 block of bytes. The number of rounds (`ROUNDS` = 10, 12, or 14) depends on the size of the key.

(2) For each 128 bit data block `d` to encrypt, perform the following operations:

(2.1) Add first round key `rk[0]` to the data block:

```
KeyAddition(d, rk[0])
```

The `KeyAddition` is performed by xor'ing the round key into the data block `d`, byte-wise.

(2.2) Perform the following operations for each intermediate round `r = 1, ..., ROUNDS-1`:

```
Substitution(d, S)
ShiftRow(d)
MixColumn(d)
KeyAddition(d, rk[r])
```

The `Substitution` step replaces each byte `b` in the data block by `S[b]`, where `S` is a so-called S-box, a 256-entry table of bytes. The S-box represents an invertible affine mapping, composed of a polynomial multiplication modulo $x^8 + 1$ followed by an addition.

The `ShiftRow` operation rotates the rows of the data block `d` left by 0, 1, 2, or 3 bytes.

The `MixColumn` operation transforms each column of the data block `d`, using polynomial multiplication modulo $x^4 + 1$.

Finally, the `KeyAddition` xors the key `rk[r]` into the data block `d`, byte-wise.

(2.3) The last round, for which `r = ROUNDS`, has no `MixColumn` operation, and therefore consists of these steps:

```
Substitution(d, S)
ShiftRow(d)
KeyAddition(d, rk[r])
```

Decryption can be performed simply by doing these steps in reverse, using the inverse `Si` of the `S`-box, inverse `MixColumn`, and so on. However, by simple algebraic properties of the algorithm, decryption can be performed by a sequence of steps very similar to that for encryption, using the round keys backwards after applying inverse `MixColumn` to the round keys `rk[1]` through `rk[ROUNDS-1]`.

4.2 Implementing AES in C#

The sequence (2) of operations must be performed for each data block to be encrypted or decrypted. It can be implemented efficiently on architectures with 32-bit words and sufficient memory, as described in Daemen and Rijmen's AES Proposal paper [16], and implemented in Cryptix's Java implementation of AES [20].

We have followed this approach, which requires auxiliary tables `T0`, `T1`, `T2`, `T3` to be built as 256-entry tables of unsigned 32-bit integers, each representing the composed action of the `Substitution` and `MixColumn` operations. Then the intermediate rounds (step 2.2) of encryption can be implemented using bitwise operations on unsigned 32-bit integers, or 4 bytes in parallel. In C# it can be done like this:

```
for(int r = 1; r < ROUNDS; r++) {
  k = rk[r];
  uint t0 =
    T0[a0 >> 24] ^
    T1[(a1 >> 16) & 0xFF] ^
    T2[(a2 >> 8) & 0xFF] ^
    T3[a3 & 0xFF] ^ k[0];
  uint t1 =
    T0[a1 >> 24] ^
    T1[(a2 >> 16) & 0xFF] ^
    T2[(a3 >> 8) & 0xFF] ^
    T3[a0 & 0xFF] ^ k[1];
  uint t2 =
    T0[a2 >> 24] ^
    T1[(a3 >> 16) & 0xFF] ^
    T2[(a0 >> 8) & 0xFF] ^
    T3[a1 & 0xFF] ^ k[2];
  uint t3 =
    T0[a3 >> 24] ^
    T1[(a0 >> 16) & 0xFF] ^
    T2[(a1 >> 8) & 0xFF] ^
    T3[a2 & 0xFF] ^ k[3];
  a0 = t0; a1 = t1; a2 = t2; a3 = t3;
}
```

Above, `k[0]` is the first column of the round key, as a 32-bit unsigned integer, `k[1]` is the second column, and so on.

4.3 A deoptimized implementation of AES

In fact, the AES middle round implementation shown above has been hand-optimized already. The most compact and general implementation is shown below, where 4-element arrays `a[]`, `t[]` and `T[]` are used instead of the variables `a`, `t` and `T` above:

```
for(int r = 1; r < ROUNDS; r++) {
  k = rk[r];
  uint[] t = new uint[4];
  for (int j = 0; j < 4; j++) {
    uint res = k[j];
    for (int i = 0; i < 4; i++)
      res ^= T[i][(a[(i + j) % 4] >> (24 - 8 * i)) & 0xFF];
    t[j] = res;
  }
  a[0] = t[0]; a[1] = t[1]; a[2] = t[2]; a[3] = t[3];
}
```

For given `i` and `j` in the range 0..3, the indexing into `T[i]` with `a[j]` can be uniformly expressed as `T[i][(a[(i+j)%4 >> (24-8*i)) & 0xFF]`. However, it is tempting to write the complicated hand-optimized form in Section 4.2 because of the presumed efficiency of hand-specializing for `i` and `j` — and indeed the resulting implementation is 5 times faster. Below we shall see that runtime code generation allows us to write the general algorithm and obtain the efficiency of the specialized one, and more.

Moreover, when `i` is 0 or 3, the shift count is 24 or 8, in which case the index expression can be simplified by eliminating the bitwise 'and' (`&`) or the shift, because `aj` is a 32-bit unsigned integer.

The first round (step 2.1, not shown) is still just a key addition (using the C# xor operator `^`), and the last round (step 2.3, not shown) uses the `S` box bytewise, since it involves no `MixColumn`. Decryption can be implemented by a similar sequence of operations, as hinted above, but when handwriting the code one will usually specialize it for efficiency, so that some of the similarity with the encryption algorithm is lost.

4.4 Runtime generation of encryption code

Using runtime code generation, one can unroll the step 2.2 `for`-loop shown above, and inline the round keys `rk[r]`. Since no computations can be performed on the basis of the round key alone, one would think that this gives no speed-up at all. However, apparently the just-in-time compiler in Microsoft's CLR can perform more optimizations on the unrolled code: it is around 35 per cent faster when encrypting or decrypting a large number of data blocks.

The three nested `for`-loops below generate code corresponding to an unrolling of the `for r`-loop shown in Section 4.2:

```
for (int r = 1; r < ROUNDS; r++) {
  k = rk[r];
  for (int j = 0; j < 4; j++) {
    ilg.Emit(OpCodes.Ldc_I4, k[j]);          // Push k[j]
    for (int i = 0; i < 4; i++) {
      ilg.Emit(OpCodes.Ldloc, T[i]);
      ilg.Emit(OpCodes.Ldloc, a[encrypt ? (i+j) % 4 : (j+4-i) % 4]);
      if (i != 3) {
        ilg.Emit(OpCodes.Ldc_I4, 24 - 8 * i);
        ilg.Emit(OpCodes.Shr_Un);
      }
      if (i != 0) {
        ilg.Emit(OpCodes.Ldc_I4, 0xFF);
        ilg.Emit(OpCodes.And);
      }
      ilg.Emit(OpCodes.Ldelem_U4);
      ilg.Emit(OpCodes.Xor);
    }
    ilg.Emit(OpCodes.Stloc, t[j]);           // Assign to tj
  }
  for (int j = 0; j < 4; j++) {              // Generate a0=t0; a1=t1; ...
    ilg.Emit(OpCodes.Ldloc, t[j]);
    ilg.Emit(OpCodes.Stloc, a[j]);
  }
}
```

Above, `k[j]` is the `j`'th column of the round key `k = rk[r]`. The variable `T[i]` holds the code generator's representation of a local variable corresponding to table `Ti` for `i` in 0..3; variable `a[j]` holds the representation of local variable `aj`; and variable `t[j]` holds the representation of local variable `tj`. The variable `encrypt` determines whether code is generated for encryption (true) or decryption (false); the tables `Ti` for encryption and decryption are different also.

One iteration of the inner `for`-loop generates code to compute `Ti[(aj >> 24-8*i) & 0xFF]` and xor it into the value on the stack top. The `if`-statements in the inner `for`-loop implement the optimization for `i` being 0 or 3, discussed in Section 4.3. Writing this logic in the code generator is actually simpler and less error-prone than hand-writing the optimized code it generates (Section 4.2).

One iteration of the middle `for`-loop generates code to compute the right-hand side in the initialization `uint ti = ...` from Section 4.2, and to assign it to `ti`. One iteration of the outer `for`-loop generates the code corresponding to one iteration of the `for`-loop in Section 4.2.

4.5 Pragmatic considerations

Approximately 256 bytes of MSIL bytecode is generated for each round of encryption, or roughly 2600 bytes of bytecode for the entire encryption function (when the key size is 128 bit). It is unclear how much x86 code is generated by the CLR just-in-time compiler from this bytecode.

Two questions are central to the practical viability of this approach to encryption:

- The encryption key equals the first round key `rk[0]` and therefore can be recovered from the generated code. Hence the generated code should not be cached anywhere on disk, and should not be accessible to other applications running in the same operating system.
- Since a specialized routine is generated for each key, it would be useful to be able to discard the generated code when it is no longer in use.

5 Sparse matrix multiplication

A plain implementation of the multiplication $R = A \cdot B$ of two $n \times n$ matrices uses n^3 scalar multiplications. When matrix B has only few non-zero elements (say, 5 percent), matrix multiplication can profitably be performed in two stages: (1) make a list of all non-zero elements in each column of B, and (2) to compute element R_{ij} of R, multiply the elements of row i of A only with the the non-zero elements of column j of B. With 100×100 matrices and 5 percent non-zero elements, this is approximately 2 times faster than plain matrix multiplication.

With runtime code generation, there is a further possibility, especially interesting if $A \cdot B$ must be computed for many different A's and a fixed sparse matrix B. Namely, split step (2) above into: (2a) for every j, generate code to compute R_{ij} for fixed j, and then (2b) use that code to compute R_{ij} for every i. Still using 100×100 matrices with 5 percent non-zero elements, this can be a further 4 to 10 times faster than sparse matrix multiplication, or 8 to 20 times faster than plain matrix multiplication.

5.1 Matrix multiplication and sparse matrix multiplication

Plain matrix multiplication $R = A \cdot B$, assuming that the matrices are non-empty rectangular arrays of appropriate sizes, can be implemented as follows:

```
final int aCols = A[0].length, rRows = R.length, rCols = R[0].length;
for (int i=0; i<rRows; i++)
  for (int j=0; j<rCols; j++) {
    double sum = 0.0;
    for (int k=0; k<aCols; k++)
      sum += A[i][k] * B[k][j];
    R[i][j] = sum;
  }
```

Sparse matrix multiplication can be implemented as shown below. We assume that the `SparseMatrix` object computed for B has a method `getCol(j)` that returns a list of the non-zero elements of the j'th column of the matrix. Each non-zero element `nz` is represented by its row `nz.k` and its value `nz.Bkj`.

```
SparseMatrix sparseB = new SparseMatrix(B);
final int rRows = R.length, rCols = R[0].length;
for (int i=0; i<rRows; i++) {
  final double[] Ai = A[i];
  final double[] Ri = R[i];
  for (int j=0; j<rCols; j++) {
    double sum = 0.0;
    Iterator iter = sparseB.getCol(j).iterator();
    while (iter.hasNext()) {
      final NonZero nz = (NonZero)iter.next();
      sum += Ai[nz.k] * nz.Bkj;
    }
    Ri[j] = sum;
  }
}
```

Note that multiplication happens in two stages as suggested above: (1) first compute the `sparseB` representation of B, then (2) multiply that with the non-sparse representation of A.

5.2 Generating a sparse multiplication routine

We now further split the second stage into two. In stage (2a) we generate code for the second stage, specialized with respect to `sparseB`, unroll the `for j`-loop and the `while`-loop, but keep the `for i`-loop. Unrolling the `for i`-loop would make only a few more operations static, and would make the generated code much larger.

The bytecode generation (stage 2a) can be implemented like this, assuming that `sparseB` is given:

```
Label loop = new Label(jvmg);
loop.define(jvmg);                          // do {
jvmg.emitLoad(varA);
jvmg.emitLoad(vari);
jvmg.emitArrayLoad(double1D_type);
jvmg.emitStore(varAi);                      // Ai = A[i]
jvmg.emitLoad(varR);
jvmg.emitLoad(vari);
jvmg.emitArrayLoad(double1D_type);
jvmg.emitStore(varRi);                      // Ri = R[i]
for (int j=0; j<B.cols; j++) {
  jvmg.emitLoad(varRi);                     // Load Ri
  jvmg.emitPushInt(j);
  jvmg.emitPushDouble(0.0);                 // sum = 0.0
  Iterator iter = B.getCol(j).iterator();
  while (iter.hasNext()) {
    final NonZero nz = (NonZero)iter.next();
    jvmg.emitPushDouble(nz.Bkj);            // load B[k][j]
    jvmg.emitLoad(varAi);                   // load A[i]
    jvmg.emitPushInt(nz.k);
    jvmg.emitArrayLoad(Type.double_type);   // load A[i][k]
    jvmg.emitMul();                         // prod = A[i][k]*B[k][j]
    jvmg.emitAdd('D');                      // sum += prod
  }
  jvmg.emitArrayStore(Type.double_type);    // R[i][j] = sum
}
jvmg.emitLoad(vari);
jvmg.emitPushInt(1);
jvmg.emitAdd('I');
jvmg.emitStore(vari);                       // i++
jvmg.emitLoad(vari);
jvmg.emitPushInt(aRows);
jvmg.emitGotoIfLt(loop);                    // } while (i<aRows);
jvmg.emitReturn();
```

Above we assume that the generated code's parameters *A* and *R* are held in generation-time variables `varA` and `varR`, and similarly for the generated code's variables `Ai`, `Ri`, and `i`.

The `i`-loop is expressed as a `do-while` loop in the generated code. The generated loop begins with the label `loop`, and ends with the conditional jump instruction generated by `emitGotoIfLt(loop)`. The use of a `do-while` loop is consistent with the assumption that the matrices are non-empty.

After the code has been generated, we use it in stage (2b) to compute all cells of the resulting matrix *R*.

5.3 Experimental results

We have implemented this idea in Java using the `gnu.bytecode` [5] bytecode generation package; see Section 2.4. Runtime generation of a sparse matrix multiplication routine specialized for B can produce code that is 4 to 10 times faster than a general sparse matrix multiplication routine, and 8 to 20 times faster than plain matrix multiplication.

These runtime figures are for the Sun HotSpot Client VM. The Sun HotSpot Server VM is faster for bytecode compiled from Java code, but (in this case) slower for bytecode generated at runtime. It is unclear why 1000 matrix multiplications take more than 15 times as long as 100 matrix multiplications in Sun HotSpot Server VM. The IBM JVM is even more variable, in that runtime code generation gives a net slowdown of 25 percent when performing 100 matrix multiplications, but a speed-up by a factor of 6 when performing 1000 matrix multiplications. This may be due to dynamic optimizations performed on frequently executed code. See Figure 5.

	100 matrix multiplications				**1000 matrix multiplications**			
	Sun HotSpot		**IBM**	**MS**	**Sun HotSpot**		**IBM**	**MS**
	Client	**Server**	**JVM**	**CLR**	**Client**	**Server**	**JVM**	**CLR**
Plain	2.749	2.302	1.067	1.432	27.489	21.890	10.535	14.230
Sparse	1.118	0.820	0.904	1.191	10.660	5.548	7.405	11.567
Sparse, 2-phase	0.993	0.458	0.684	0.931	9.814	4.438	6.737	9.263
Sparse, rtcg	0.222	0.467	1.341	0.300	0.920	6.995	1.482	0.691

Figure 5: Time in seconds for multiplications of 100×100 sparse (5 percent) matrices.

The expected number of instructions in the specialized multiplication method for 100×100 sparse (5 percent) matrices is $100 \cdot 5 \cdot 15$ bytes, that is, 37.5 KB. This seems to translate into 60 KB of generated assembly code and other overhead when compiled by the just-in-time compiler in Sun HotSpot Client VM, and 70 KB in MS CLR; but in general we have no accurate way to estimate this number. The time to generate a specialized sparse multiplication method is roughly 15 ms with Sun HotSpot Client VM, and roughly 260 ms with MS CLR. For Sun HotSpot Client VM, runtime code generation pays off after only two uses of the generated function, whereas 40 uses are required on the MS CLR platform.

6 Conclusion

We have demonstrated that runtime code generation is well supported by modern execution platforms such as Sun's HotSpot JVM, IBM's JVM, and Microsoft's Common Language Runtime. This is due primarily to:

- the simplicity of generating stack-oriented portable bytecode,
- the support from bytecode verifiers, and
- highly optimizing just-in-time native code generators.

Together these features make portable runtime code generation fairly easy and safe, and the generated code efficient. We demonstrated this using several small examples, and two slightly larger ones.

We have found the Microsoft CLR runtime code generation facilities to be well-documented and well integrated with the reflection facilities.

For the Java Virtual Machine one needs to use third-party code generation libraries. The `gnu.bytecode` runtime code generation facilities are rather poorly documented, but they are well-designed and therefore fairly easy to use. The BCEL runtime code generation facilities are fairly well-documented, but offers many ways to do the same thing, and may be somewhat bewildering for this reason.

First, we have shown how to implement delegates (as known from C#) in Java using runtime code generation. Using delegates one can avoid some of the inefficiency usually incurred by reflective method calls. This works better with the Sun HotSpot Client VM than with the IBM JVM.

Secondly, we have shown that the Advanced Encryption Standard (AES, Rijndael) algorithm can be implemented efficiently in C# using runtime code generation, reaching speeds otherwise not possible on the managed Common Language Runtime platform.

Third, we have shown that runtime code generation can be profitably applied also to sparse matrix multiplication, at least when the matrices are large enough and sparse enough.

The execution platforms (Sun HotSpot Client JVM, Sun HotSpot Server JVM, IBM JVM, and Microsoft's CLR) display quite diverging execution speeds for different types of programs. For instance, the IBM JVM executes bytecode compiled from Java program faster than Sun HotSpot Client VM does, but is considerably slower when executing reflective method calls and when executing delegates as implemented in Section 3. The Sun HotSpot Server VM appear to represent a middle point between these. The Microsoft CLR is comparable to IBM JVM when executing bytecode compiled from Java, and executes code generated at runtime very fast, too.

In general, the execution times of JIT-compiled code are subject to considerable variations, probably because of the dynamic and somewhat heuristic nature of the optimizations performed by the JIT-compiler. Often, the execution times exhibit strange but reproducible variations between very similar bytecode programs.

Moreover, the hardware platform (e.g. Intel Pentium 3 versus AMD Athlon) affects code speed in non-obvious ways. In particular, the speed-up factor obtained by runtime code generation can be very different on the two architectures.

We have focused mainly on technological aspects of runtime code generation. To make runtime code generation more practical and safer, research could focus on these aspects:

- Language mechanisms that permit the programmer to work at the Java or C# level instead of the bytecode level. Previous work in this direction is represented by Tick C [19], and MetaML [33] MetaOCaml [10]. The DynJava implementation [31], and Cisternino and Kennedy's C# toolkit [11] seem to be the only attempts in this direction for Java and C#.

- A code generation framework or type system that could guarantee at Java or C# compiletime that the generated bytecode will pass the verifier. Although the JVM or CLR verifier will catch all code errors, it is far better to have a static guarantee that the generated code will be verifiable. Previous relevant work includes that by Davies and Pfenning [17], and the type systems of MetaOCaml [10] and DynJava [31].

Acknowledgements: Martin Elsman suggested looking at encryption algorithms for a case study, and provided comments on a draft. Thanks also to Kasper Østerbye, Ken Friis Larsen, and Niels Jørgen Kokholm for pointers and suggestions.

References

[1] Bytecode engineering library. At `http://jakarta.apache.org/bcel/`.

[2] Gnu lightning homepage. At `http://www.gnu.org/software/lightning/`.

[3] Mono project homepage. At `http://www.go-mono.com/`.

[4] Open runtime platform homepage. At `http://orp.sourceforge.net/`.

[5] `gnu.bytecode` bytecode generation tools. At `http://www.gnu.org/software/kawa/`.

[6] The Java HotSpot virtual machine. Technical white paper, Sun Microsystems, 2001. At `http://java.sun.com/docs/white/`.

[7] Specification for the Advanced Encryption Standard. Federal Information Processing Standards Publication 197, National Institute of Standards and Technology, USA, 2001. At `http://csrc.nist.gov/publications/`.

[8] J. Auslander et al. Fast, effective dynamic compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 149–158, May 1996.

[9] T.M. Breuel. Implementing dynamic language features in Java using dynamic code generation. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, TOOLS 39*, pages 143–152, 2001.

[10] C. Calcagno, W. Taha, L. Huang, and X. Leroy. A bytecode-compiled, type-safe, multi-stage language. 2001. At `http://www.dcs.qmw.ac.uk/~ccris/ftp/pldi02-pre.pdf`.

[11] Antonio Cisternino and Andrew Kennedy. Language independent program generation. University of Pisa and Microsoft Research Cambridge UK, 2002.

[12] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL'96: 23rd Principles of Programming Languages, St. Petersburg Beach, Florida, January 1996*, pages 145–156, 1996.

[13] IBM Corporation. Ibm java developer kit. At `http://www-106.ibm.com/developerworks/java/jdk/`.

[14] IBM Corporation. Jikes research virtual machine (rvm). At `http://www-124.ibm.com/developerworks/oss/jikesrvm/`.

[15] K. Czarnecki and U. W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000.

[16] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. Technical report, Proton World Intl. and Katholieke Universiteit Leuven, Belgium, 1999. At `http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf`.

[17] R. Davies and F. Pfenning. A modal analysis of staged computation. In *23rd Principles of Programming Languages, St Petersburg Beach, Florida*, pages 258–270. ACM Press, 1996.

[18] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Programming Language Design and Implementation*, 1996. At `http://www.pdos.lcs.mit.edu/~engler/vcode-pldi.ps`.

[19] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: a language for high-level, fast dynamic code generation. In *23rd Principles of Programming Languages, St Petersburg Beach, Florida*, pages 131–144. ACM Press, 1996. At `http://www.pdos.lcs.mit.edu/~engler/pldi-tickc.ps`.

[20] Cryptix Foundation. Cryptix aes kit. At `http://www.cryptix.org/products/aes/`.

[21] Brian Gladman. AES algorithm efficiency. At `http://fp.gladman.plus.com/cryptography_technology/aes/`.

[22] B. Grant et al. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1-2):147–199, October 2000. Also `ftp://ftp.cs.washington.edu/tr/1997/03/UW-CSE-97-03-03.PS.Z`.

[23] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[24] U. Jørring and W.L. Scherlis. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 86–96. New York: ACM, 1986.

[25] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Enginnering, University of Washington, November 1991. At `ftp://ftp.cs.washington.edu/tr/1991/11/UW-CSE-91-11-04.PS.Z`.

[26] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimisations. Technical Report UW-CSE-93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993. At `ftp://ftp.cs.washington.edu/tr/1993/11/UW-CSE-93-11-02.PS.Z`.

[27] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Programming Language Design and Implementation*, pages 137–148, 1996.

[28] D. Lomov and A. Moskal. Dynamic Caml v. 0.2. run-time code generation library for objective caml. Technical report, Sankt PetersBurg State University, Russia, May 2002. At `http://oops.tercom.ru/dml/`.

[29] Sun Microsystems. Java 2 platform, standard edition. At `http://java.sun.com/j2se/`.

[30] F. Nielson. A formal type system for comparing partial evaluators. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 349–384. North-Holland, 1988.

[31] Y. Oiwa, H. Masuhara, and A. Yonezawa. Dynjava: Type safe dynamic code generation in java. In *JSSST Workshop on Programming and Programming Languages, PPL2001, March 2001, Tokyo*, 2001. At `http://wwwfun.kurims.kyoto-u.ac.jp/ppl2001/papers/Oiwa.pdf`.

[32] Morten Rhiger. Compiling embedded programs to byte code. In S. Krishnamurthi and C.R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL'02). Lecture Notes in Computer Science, vol. 2257*, pages 120–136. Springer-Verlag.

[33] T. Sheard. Using MetaML: A staged programming language. In S.D. Swierstra, P.R. Henriques, and José N. Oliveira, editors, *Summer School on Advanced Functional Programming, Braga, Portugal, September 1998. Lecture Notes in Computer Science, vol. 1608*, pages 207–239. Springer-Verlag, 1999. At `http://www.cse.ogi.edu/~sheard/papers/summerschool.ps`.

[34] P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and modal-ML. In *Programming Language Design and Implementation*, pages 224–235, 1998.