Mainstream execution platforms: The Java Virtual Machine (JVM) and Common Language Runtime (CLR)

At compiletime, Java and C# are compiled to bytecode for a stack machine.

At runtime, the bytecode is compiled to real machine code by a just-in-time compiler (JIT).

Runtime code generation (RTCG) in JVM and CLR

The runtime systems support reflection — more similar to Scheme/Smalltalk than to C/C++.

Moreover, at runtime new bytecode can be generated and loaded, and the JIT will compile it to machine code.

Same purpose as compile-time specialization: Generate specialized fast code once, use it many times.

Runtime code generation: Some research topics

- Typesafety and nice type systems for multi-stage languages [not this talk].
- Language notations for convenient code generation [not this talk].

This talk: Runtime code generation has lots of practical potential

- Traditionally there has been a conflict between:
  - Portability: Generate bytecode; slow, and closed world, separate from mainstream software.
  - Speed: Generate machine code; non-portable, and too much engineering effort to maintain.
- Claim: JVM and CLR provide portability and speed.
  Because of bytecode plus JIT plus lots of engineering effort, made by others.
- We show that bytecode can be very fast, and not to great a pain to generate.
- Case study: the Advanced Encryption Standard (AES, Rijndael) in C#.
- Case study: Fast sparse matrix multiplication in Java.
- The need for better tools: type systems and notations.

The Java Virtual Machine (JVM)

The JVM is a specification of an abstract machine. Some implementations:

- Sun HotSpot Client VM, 'fast JIT, slow code': java -client
- Sun HotSpot Server VM, 'slow JIT, fast code': java -server
- IBM JIT JVM, 'slow JIT, fast code'

Runtime code generation kit: gnu.bytecode; others include Apache BCEL and ASM.

The Common Language Runtime (CLR) for C#, VB.Net, …

The CLR is a specification of an abstract machine. Some implementations:

- The Mono project's CLR 1.1.9

Runtime code generation kit: namespace System.Reflection.Emit

Both platforms are widely used: desktop and server installations are counted in tens or hundreds of millions.
Initial question: Is bytecode generated at runtime fast enough to bother?

Initial answer: Yes, surprisingly fast

Consider a trivial loop:
```java
do {
    n--;  
} while (n != 0);
```

A generated version of the loop expressed in CLR bytecode:
```java
start: Ldarg_0  
Ldc_I4_1  
Sub  
Starg_0  
Ldarg_0  
Brtrue start
```

The machine code JIT'ed from the bytecode (by Mono for x86) is:
```
12: dec%edi
13: test%edi,%edi
15: jne12
```

Generated bytecode is as fast as compiled Java and C# and even C (1.6 GHz Pentium M):

|              | Sun HotSpot | IBM | MS | Mono | gnu|g
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>392</td>
<td>∞</td>
<td>781</td>
<td>775</td>
<td>775</td>
</tr>
<tr>
<td>Server</td>
<td>392</td>
<td>781</td>
<td>775</td>
<td>775</td>
<td>515</td>
</tr>
</tbody>
</table>

Iterations per second: bigger is better.

Generated bytecode is faster than C (2.8 GHz Pentium 4): Long pipeline, alignment artefacts?

|              | Sun HotSpot | IBM | MS | Mono | gnu|g
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>875</td>
<td>∞</td>
<td>1770</td>
<td>2220</td>
<td>1727</td>
</tr>
<tr>
<td>Server</td>
<td>875</td>
<td>1770</td>
<td>2220</td>
<td>1755</td>
<td>1818</td>
</tr>
</tbody>
</table>

Iterations per second: bigger is better.

What does runtime bytecode generation look like? Example: Evaluation of polynomials

The polynomial $p(x)$ of degree $n$ with coefficient array $cs[0...n]$:

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

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

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

Given coefficient array $cs[\cdot]$ and $x$, we can compute $p(x)$ with result in variable res:

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

Potential for staging, or splitting of binding-times:

If a given polynomial $p(x)$ must be evaluated for many different values of $x$, then do it in two stages:

1. Generate specialized code for the given coefficient array $cs[\cdot]$; then
2. For every $x$, execute the specialized code.

The specialized code for a given polynomial

Let constant $cs_i$ be the value of $cs[i]$: 

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

The corresponding stack-oriented bytecode for CLR

```java
LdcR8 0.0 // push res = 0.0 on stack  
Ldarg_0 // load x  
Mul // compute res * x  
LdcR8 cs_n // load cs[n]  
Add // compute res * x + cs[n]  
...  
Ldarg_0 // load x  
Mul // compute res * x  
LdcR8 cs_0 // load cs[0]  
Add // compute res * x + cs[0]  
Return // return res
```

Generated bytecode is as fast as compiled Java and C# and even C (1.6 GHz Pentium M)
How to generate the specialized code in C#/CLR

The standard evaluation loop:

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

To generate bytecode, one uses a bytecode generator `ilg` of type `ILGenerator`:

```csharp
  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_0); // 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;
```

Further optimization: skip term if coefficient \(cs[i]\) is zero

In the standard evaluation loop the optimization makes no sense; an addition is likely faster than a test:

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

But in the polynomial evaluator generator, the test can be performed once, at bytecode generation time:

```csharp
  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_0); // 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;
```

The generated code is faster only if the coefficient array \(cs[i]\) is long \((\geq 10)\) or many coefficients are zero.

For more complicated expressions the speed-up is larger. Spreadsheets, graphing applications, ...

---

A surprising concern: How do we get to execute the generated code?

Bytecode must belong to a method; to create and execute bytecode we must create and call a method.

- Approach 1: A method must belong to a class; a class to a module; and a module to an assembly.
  So create an AssemblyBuilder, a ModuleBuilder, a TypeBuilder, a MethodBuilder, and an ILGenerator.
  Use to generate bytecode in a method in a class implementing a given interface. Create instance of class and cast to the interface, then call generated method by an interface call.

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

- Approach 2 (since CLR 2.0): Use class DynamicMethod to add a new static method to an existing module.
  But a DynamicMethod must be called as a delegate, and that is surprisingly slow:

<table>
<thead>
<tr>
<th>Call to generated</th>
<th>Time (ns/call)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface call</td>
<td>10</td>
</tr>
<tr>
<td>Delegate call</td>
<td>46</td>
</tr>
<tr>
<td>Reflective call</td>
<td>2834</td>
</tr>
</tbody>
</table>

- Approach 3: Same as approach 1, but use generic types and reflection to hide the gory details.

---

Intergalactic phrasebook: Function types and function values in C# and Java

<table>
<thead>
<tr>
<th>C# delegate type and anonymous method</th>
<th>Standard ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>public delegate R D(A x);</td>
<td>type D = A -&gt; R</td>
</tr>
<tr>
<td>delegate(A x) { return e; }</td>
<td>(\text{fn}(x : A) = e)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C# and Java interface types</th>
<th>Standard ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>public interface IFun {</td>
<td>type IFun = { Invoke : A -&gt; R }</td>
</tr>
<tr>
<td>R Invoke(A x);</td>
<td>}</td>
</tr>
<tr>
<td>public interface IFun&lt;A,R&gt;</td>
<td>type (&quot;A&quot;, &quot;R&quot;) IFun = { Invoke : 'A -&gt; 'R }</td>
</tr>
<tr>
<td>R Invoke(A x);</td>
<td>}</td>
</tr>
</tbody>
</table>
**Approach 1:** Generating a virtual method in a class implementing an interface, as if declared like this:

```csharp
class MyClass : IMyInterface {
    public virtual double MyMethod(double x) { return x + 2.1; }
}
```

Generate assembly, module, and class method; create instance, cast to interface, and call method:

```csharp
AssemblyName assemblyName = new AssemblyName();
assemblyName.Name = "myassembly";
AssemblyBuilder assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
ModuleBuilder moduleBuilder = assemblyBuilder.DefineDynamicModule("mymodule");
TypeBuilder typeBuilder = moduleBuilder.DefineType("MyClass", TypeAttributes.Class | TypeAttributes.Public);
{ConstructorBuilder constructorBuilder = typeBuilder.DefineConstructor(MethodAttributes.Public, CallingConvention.HasThis, new Type[]{});
    ILGenerator ilg = constructorBuilder.GetILGenerator();
    ilg.Emit(OpCodes.Ldarg_0);
    ilg.Emit(OpCodes.Call, typeof(Object).GetConstructor(new Type[]{()}));
    ilg.Emit(OpCodes.Ret);
}
{MethodBuilder methodBuilder = typeBuilder.DefineMethod("MyMethod", MethodAttributes.Virtual | MethodAttributes.Public, typeof(double), new Type[]{typeof(double)});
    ILGenerator ilg = methodBuilder.GetILGenerator();
    ilg.Emit(OpCodes.Ldarg_1);
    ilg.Emit(OpCodes.Ldc_R8, 2.1);
    ilg.Emit(OpCodes.Add);
    ilg.Emit(OpCodes.Ret);
} Type ty = typeBuilder.CreateType();
Object obj = ty.GetConstructor(new Type[]{()}).Invoke(new Object[]{()});
IMyInterface mm = (IMyInterface)obj;
mm.MyMethod(5.7);
```

**Approach 2:** Generating a static method in an existing module, as if declared like this:

```csharp
public static double MyMethod(double x) { return x + 2.1; }
```

The generated method can be wrapped as a delegate of type D2D:

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

Generate dynamic method, create delegate, and call it:

```csharp
DynamicMethod methodBuilder = new DynamicMethod("MyMethod", typeof(double), new Type[]{ typeof(double) }, typeof(String).Module);
ILGenerator ilg = methodBuilder.GetILGenerator();
ilg.Emit(OpCodes.Ldarg_0);
ilg.Emit(OpCodes.Ldc_R8, 2.1);
ilg.Emit(OpCodes.Add);
ilg.Emit(OpCodes.Ret);
D2D mm = (D2D)methodBuilder.CreateDelegate(typeof(D2D));
mm(5.7);
```

**Approach 3:** Generating virtual method in class implementing generic interface instance:

We want a method, as if declared like this:

```csharp
class MyClass : IFun<double,double> {
    public virtual double Invoke(double x) { return x + 2.1; }
}
```

The generic interface IFun<A,R> describes functions of type A -> R:

```csharp
public interface IFun<A,R> {
    R Invoke(A x);
}
```

Generate class and method; create instance, cast to interface, and call method. An anonymous delegate generates the method body:

```csharp
IFun<double,double> mm = RtcgFun<double,double>(delegate (ILGenerator ilg) {
    ilg.Emit(OpCodes.Ldarg_1);
    ilg.Emit(OpCodes.Ldc_R8, 2.1);
    ilg.Emit(OpCodes.Add);
    ilg.Emit(OpCodes.Ret);
});
mm.Invoke(5.7);
```

**The implementation of approach 3**

Method `MethodObject encapsulates most of boilerplate from approach 1.`

```csharp
static Object MethodObject (ILFiller ilFiller, Type iface, Type rTy, params Type[] argTy) {...}
```

The method makes a class as if declared like this:

```csharp
public class ClassK : iface {
    public virtual rTy Invoke(argTy) { ... }
}
```

and then creates and returns an instance of it.

```csharp
Method RtcgFun<A,R> calls MethodObject and casts the resulting object to a suitable interface:

```csharp
public static IFun<A,R> RtcgFun<A,R>(ILFiller ilFiller) {
    return (IFun<A,R>) MethodObject (ilFiller, typeof(IFun<A,R>), typeof(R), typeof(A));
}
```

This is typesafe and makes essential use of reflection on generic type instances.

This works in C#/.NET, but not in Java/JVM due its ‘generics by erasure’ implementation.
Case study: The Advanced Encryption Standard (AES, Rijndael)

US Federal standard for sensitive information, since May 2002.

AES is a block cipher with 128-bit blocks, and key size 128, 192, or 256 bit.

(1) Given a key, generate an array \( r_k[0..\text{ROUNDS}] \) of round keys, where \( \text{ROUNDS} = 10, 12, \) or \( 14 \).

(2) For each 128-bit data block \( d \) to encrypt, do:

(2.1) Xor first round key \( r_k[0] \) into the data block \( d \).

(2.2) For the middle rounds \( r = 1..\text{ROUNDS}-1 \) do:

- Substitution \( (d, S) \) // \( S \) defines an invertible affine mapping
- ShiftRow \( (d) \) // rotate each row by a different amount
- MixColumn \( (d) \) // transform columns by polynomial mult.
- KeyAddition \( (d, r_k[r]) \) // xor round key \( r_k[r] \) into the data block

(2.3) The last round, with \( r = \text{ROUNDS} \), is like (2.2) but has no MixColumn.

All operations can be implemented using bitwise operations (shift, xor, or), and some auxiliary tables.

Potential for staging:

Step (1) is performed once for a given key, and step (2) is performed for each data block: many times.

---

A direct but naïve implementation of the AES middle rounds (2.2)

The 128-bit data block to encrypt is in \( a[0..3] \), a four-element array of 32-bit unsigned integers.

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

The round keys are in array \( r_k[0..\text{ROUNDS}] \).

The arrays \( T[0..3] \) are precomputed from table \( S \).

---

Hand-optimized implementation of the AES middle rounds (AES submission)

```
for (int r = 1; r < \text{ROUNDS}; r++) {
    k = r_k[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]);
            if (i != 0) ilg.Emit(OpCodes.Ldloc, a[(i + j) & 4]);
            ilg.Emit(OpCodes.Ldelem_U4);
            ilg.Emit(OpCodes.Stloc, t[j]); // Assign to t[j]
        }
    }
}
```

The inner loops have been unrolled, the data block array \( a[0..3] \) replaced by variables \( a0, ..., a3 \), and 2D table \( T[0..3] \) has been replaced by 1D tables \( T0, ..., T3 \).

---

A runtime code generator for optimized AES

```
for (int r = 1; r < \text{ROUNDS}; r++) {
    k = r_k[r];
    for (int j = 0; j < 4; j++) {
        ilg.Emit(OpCodes.Ldc_I4, j);
        for (int i = 0; i < 4; i++) {
            ilg.Emit(OpCodes.Ldloc, T[i]);
            if (i != 0) ilg.Emit(OpCodes.Ldloc, a[(i + j) & 4]);
            ilg.Emit(OpCodes.Ldelem_U4);
            ilg.Emit(OpCodes.Stloc, t[j]); // Assign to t[j]
        }
    }
}
```

Structure is similar that of naïve algorithm, but performance better than the hand-optimized one.
Performance of specialized AES

No computations depend only on the round key $r[k][r]$, so one should expect very little speed-up.

But improved instruction scheduling and pipeline usage (?) do give a speed-up:

<table>
<thead>
<tr>
<th>1.6 Pentium M</th>
<th>2.8 Pentium 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand-optimized</td>
<td>201 200 327 320</td>
</tr>
<tr>
<td>Specialized (RTCG)</td>
<td>260 255 465 386</td>
</tr>
</tbody>
</table>

Encryption speed in Mbit/s: Bigger is better.

(MS CLR 2.0 beta 2 on Windows 2000 and Mono 1.1.9 on Linux; 1.6 GHz P M and 2.8 GHz P 4).

The use of DynamicMethod would give a performance loss of 7.5%.

Best commercial C implementation for 1.6 GHz Pentium M encrypts ca. 560 Mbit/s.

Approximately 3 KB of bytecode is generated for each encryption key.

Bytecode generation and just-in-time compilation takes less than 10 ms and ca. 12.7 KB space for each key.

Sparse matrix multiplication

Plain multiplication $R = A \cdot B$ of two $n \times n$ matrices uses $n^3$ scalar multiplications:

```java
for (int i=0; i<nRows; i++) {
    double sum = 0.0;
    for (int j=0; j<nCols; j++) {
        sum += A[i][k] * B[k][j];
        R[i][j] = sum;
    }
}
```

If $B$ has few non-zero elements, we can find the non-zeroes of each $B$ column and then multiply with $A$'s rows.

Assume we need to compute $A \cdot B$ for fixed sparse $B$ and many different non-sparse $A$.

Potential for staging:

1. Given $B$, generate specialized $B$-multiplier; 2. apply to all relevant $A$ matrices.

Unrolling the $j$ loop is essential because it selects the columns of the given $B$.

Unrolling the $i$ loop would generate much more code and offers little extra benefit; so don’t do it.

Severe usability problems: Simple (type) mistakes are hard to debug

- MS CLR, applying instruction to too few arguments:
  Unhandled Exception: System.InvalidProgramException:
  Common Language Runtime detected an invalid program.
  at MyClass.MyMethod(Double)
  at RTCG2.Main(String[] args)

- MS CLR, adding integer to double:
  No message, just a meaningless result.

- MS CLR, another typical error message:
  Operation could destabilize the runtime.

- GNU Lightning, using the wrong macro:
  `jit_addr_i(JIT_R0, JIT_R0, JIT_R1);` // Effect: R0 = R0 + R1
  `jit_addi_i(JIT_R0, JIT_R0, JIT_R1);` // Effect: R0 = R0 + 66

- Sun Hotspot JVM with gnu.bytecode, applying instruction to too few arguments; almost helpful:
  Exception in thread "main" java.lang.Error:
  popType called with empty stack MyClass.MyMethod(double)double
Conclusions so far

- The JVM and CLR are interesting, widely available platforms supporting runtime code generation.
- Bytecode generation plus JIT compilation provides for portability and fast generated code.
- The performance of JIT-based implementations is somewhat unpredictable, due to adaptive optimizations.

Related work

- Lisp/Scheme quasiquotation with \('\) and \(\{\cdot\}\) and eval from MIT Lisp, ca. 1978.
- Untyped two-level C: Tick C (Engler et al. 1996).
- Somewhat portable runtime code generation in C: GNU Lightning, seriously untyped.
- Runtime specialization in Tempo (Consel).

How to make RTGC on JVM and CLR more usable to average programmers?

- Better source language syntax: multi-stage versions of Java and C#:
  ```
  \{'( double res = 0.0; )
  for (int i = cs.length-1; i >= 0; i--)
  \{'( res = res * x + $cs[i]; )
  \{'( return res; )
  ```
  Some languages extensions with quasiquotation exist: DynJava (Oiwa 2001); MetaPhor/C# (Neverov and Roe 2004); Genoupe/C# (Draheim 2005); SafeGen/Java (Smaragdakis 2005).

- Other languages on same platforms. Multi-stage F#?
- Much needed: Type systems to help catch errors.
  Ideally: guarantee (1) sensible binding-times, (2) correct structure and (3) correct types of generated code.
  MetaOCaml, Genoupe, SafeGen … represent work in that direction
- Simpler and better support for calling, and then discarding, generated code.
- Security: who can access and read the code I generate?