Runtime code generation
in JVM and .Net CLR *

Peter Sestoft (sestoft@dina.kvl.dk)
Royal Veterinary and Agricultural University
and
IT University of Copenhagen, Denmark

The Java Virtual Machine (JVM) and Microsoft’s .Net Common Language Runtime (CLR)
At compile-time, Java is compiled to bytecode for a stack machine, the Java Virtual Machine.
At run-time, the bytecode is compiled to real machine code by a just-in-time compiler (JIT).

Similarly, Microsoft’s C# is compiled to bytecode, which is JIT-compiled to machine code for CLR.

Runtime code generation (RTCG) in JVM and CLR
At runtime new bytecode can be generated and loaded, and the JIT will compile it to machine code.
One purpose is to generate specialized, faster code.
Useful for adapting algorithms to data that become available only at runtime.

This talk
● Explain runtime code generation in Java and C# by examples.
● Give a quantitative assessment of current technology.
● Show that bytecode makes runtime code generation fairly simple and portable.
● Show that generated code can be fast, thanks to just-in-time compilers.
● Case study: Fast implementation of the Advanced Encryption Standard (AES, Rijndael) in C#.
● Case study: Fast sparse matrix multiplication in Java and C#.
● (Case study: Efficient reflective method calls in Java via delegates).

The lecture slides and examples (and a paper, soon) are available at:
http://www.dina.kvl.dk/~sestoft/rtcg/

The Java Virtual Machine (JVM)
The JVM is a specification of an abstract machine. There are several implementations:
● Sun HotSpot Client VM, ‘fast JIT, slow code’, also known as java -client
● Sun HotSpot Server VM, ‘slow JIT, fast code’, also known as java -server
● IBM JIT JVM, ‘slow JIT, fast code’
● IBM Research JVM (RVM), a platform for experiments with JIT-compilation
● … others
Runtime code generation kits: gnu.bytecode and BCEL (Bytecode Engineering Library).

The Common Language Runtime (CLR)
The CLR is a specification of an abstract machine. There are several implementations:
● Microsoft’s .Net CLR 1.0
● Microsoft’s shared source implementation of CLR, also known as Rotor
● The Mono project’s CLR (runtime code generation facilities currently incomplete)
Runtime code generation kit: namespace System.Reflection.Emit
No speed penalty for code generated at runtime; and runtime code generation is fast

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

The loop is equally fast, whether compiled from Java/C# or generated at runtime:

<table>
<thead>
<tr>
<th>Language</th>
<th>Client</th>
<th>Server</th>
<th>JVM</th>
<th>CLR</th>
<th>gcc-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun HotSpot 1.4.0</td>
<td>∞</td>
<td>421</td>
<td>408</td>
<td>422</td>
<td></td>
</tr>
<tr>
<td>IBM 1.3.1</td>
<td>243</td>
<td>421</td>
<td>408</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>MS Net CLR 1.0</td>
<td>243</td>
<td>421</td>
<td>408</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>200</td>
<td>142</td>
<td>180</td>
<td>100</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Sun HotSpot 1.4.0 and IBM JIT 1.3.1 for Linux, and MS .Net CLR 1.0 on Windows 2000 under VmWare.

Hardware: 850 MHz Pentium 3.

Stack operations upset MS CLR: using Load; Dup instead of Load; Load can make code 37 % slower!

Runtime code generation example: Evaluation of polynomials

The polynomial \( p(x) \) of degree \( n \) with coefficient array \( c_0 \ldots c_n \):

\[
p(x) = c_0 \cdot x^n + c_1 \cdot x^{n-1} + \cdots + c_n \cdot x^0
\]

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

\[
p(x) = c_0 + x \cdot (c_1 + x \cdot (\cdots + x \cdot (c_n \cdot x^n + c_{n-1} + \cdots + x \cdot c_1 + c_0))
\]

Given coefficient array \( c_0 \ldots c_n \) and \( x \), we can compute \( p(x) \) with result in variable \( res \):

```c
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 the 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 \( c_0 \ldots c_n \); then (2) for every \( x \), execute the specialized code.

The specialized code for a given polynomial

The constant \( c_{-i} \) is the value of \( c[i] \):

```c
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)

- \( Ldc_R8 \ 0.0 \) // push res = 0.0 on stack
- \( Ldarg_0 \) // load x
- \( Mul \) // compute res * x
- \( Ldc_R8 \ cs_n \) // load cs[n]
- \( Add \) // compute res * x + cs[n]
- \( Ldarg_0 \) // load x
- \( Mul \) // compute res * x
- \( Ldc_R8 \ cs_0 \) // load cs[0]
- \( Add \) // compute res * x + cs[0]
- \( Return \) // return res

How to generate the specialized code in C#/CLR

To generate bytecode, one uses a bytecode generator \( ilg \):

```c
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

```c
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 \( c_0 \ldots c_n \) is long (\( \geq 20 \)) or many coefficients are zero.
The power example

Computing \( x^n \), that is, \( x \) to the \( n \)’th power:

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

Staging: When \( n \) is known, the computations involving \( n \) can be performed and the `for`-loop can be unrolled.

For \( n = 7 \), this gives:

```java
int p;
p = 1;
p = p * x;
x = x * x;
p = p * x;
x = x * x;
p = p * x;
return p;
```

Generating the function \( \text{power}(x, n) \) for a fixed \( n \)

```java
public static void PowerGen(ILGenerator ilg, int n) {
    ilg.DeclareLocal(typeof(int)); // declare p as local_0
    ilg.Emit(OpCodes.Ldc_I4_1); // p = 1;
    while (n > 0) {
        if (n & 2 == 0) {
            ilg.Emit(OpCodes.Ldarg_0); // x is arg_0 in generated method
            ilg.Emit(OpCodes.Ldarg_0);
            ilg.Emit(OpCodes.Mul);
            ilg.Emit(OpCodes.Starg_S, 0); // x = x * x
            n = n / 2;
        } else {
            ilg.Emit(OpCodes.Ldloc_0);
            ilg.Emit(OpCodes.Ldarg_0);
            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;
}
```

For \( n = 16 \), the specialized code is 35 percent faster than the general code.

The full story: runtime code generation in the CLR

The bytecode generator \( \text{ilg} \) generates a method body. A method must belong to a class. A class must belong to a module. A module must belong to an assembly.

So one needs an AssemblyBuilder, a ModuleBuilder, a TypeBuilder, a MethodBuilder, and an ILEGenerator.

These classes are defined in the System.Reflection.Emit namespace of the .Net Framework.

These are the steps needed to generate a method `MyClass.MyMethod`:

- `AssemblyName assemblyName = new AssemblyName();`
- `AssemblyBuilder assemblyBuilder = ... assemblyName ... ModuleBuilder moduleBuilder = ... moduleBuilder.DefineDynamicModule( ...) TypeBuilder typeBuilder = ... typeBuilder.CreateType(...) ILGenerator ilg = ... GetILGenerator(...) ... use ilg to generate the body of method MyClass.MyMethod ... Type ty = ... CreateType()`

To call the generated method `MyClass.MyMethod`, obtain a `MethodInfo` object by reflection, and call it:

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

The Advanced Encryption Standard (AES, Rijndael)

US Federal standard for sensitive (unclassified) 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 `rk[0..ROUNDS]` of round keys, where `ROUNDS = 10, 12, or 14`.
2. For each 128-bit data block \( d \) to encrypt, do:
   2.1) XOR first round key \( rk[0] \) into the data block \( d \).
   2.2) For the middle rounds \( r = 1..ROUNDS-1 \) 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`, `rk[r]`) \( \) XOR round key `rk[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 somewhat naïve implementation of the AES middle rounds (2.2)

```c
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;
    }
}
```

The round keys are in array rk[0..ROUNDS].
The data block to encrypt is in a[0..3], an array of 32-bit unsigned integers.
The arrays T[0..3] are derived from table S.

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

```c
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[(i+j) % 4]);
            if (i != 3) ilg.Emit(OpCodes.Ldc_I4, 24-8*i); // Shift
            if (i != 0) ilg.Emit(OpCodes.Ldc_I4, 0xFF); // AND
        ilg.Emit(OpCodes.Xor);
    }
    for (int j = 0; j < 4; j++) { // Generate a0=t0; a1=t1; ...
        ilg.Emit(OpCodes.Stloc, t[j]); // Assign to t[j]
    }
}
```

The data block to encrypt is in a0,..., a3, corresponding to a[0..3] before.
Tables T0,..., T3 correspond to T[0..3] before.

A runtime code generator for optimized AES

```c
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[(i+j) % 4]);
            if (i != 3) ilg.Emit(OpCodes.Ldc_I4, 24-8*i); // Shift
            if (i != 0) ilg.Emit(OpCodes.Ldc_I4, 0xFF); // AND
        ilg.Emit(OpCodes.Xor);
    }
    for (int j = 0; j < 4; j++) { // Generate a0=t0; a1=t1; ...
        ilg.Emit(OpCodes.Stloc, t[j]); // Assign to t[j]
    }
}
```

Performance

<table>
<thead>
<tr>
<th></th>
<th>Encryption (Mbit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Somewhat naïve</td>
<td>19.1</td>
</tr>
<tr>
<td>Hand-optimized</td>
<td>103.1</td>
</tr>
<tr>
<td>Specialized</td>
<td>133.1</td>
</tr>
</tbody>
</table>

Measured on MS.Net CLR 1.0 SP2 on Windows 2000 under VmWare 3 under Linux; 850 MHz Pentium 3.
Native Pentium 3 rotate instructions (unavailable in C, C++, C#) should give 280–300 Mbit/s.
Approximately 3 KB of bytecode is generated for each encryption key.
Bytecode generation and just-in-time compilation takes approx. 13 ms and 12.7 KB space for each encryption 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<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;
    }
```

If \( B \) has few non-zero elements, we can find the non-zeroes of each row and then multiply with \( A \)'s columns:

```java
SparseMatrix sparseB = new SparseMatrix(B);
for (int i=0; i<rRows; i++) {
    final double[] Ai = A[i], 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;
    }
}
```

What if we need to compute \( A \cdot B \) for fixed \( A \) and many different \( B \)?

Performance of sparse matrix multiplication (100 \( \times \) 100 matrices, 5% non-zeroes)

<table>
<thead>
<tr>
<th></th>
<th>100 matrix multiplications</th>
<th>1000 matrix multiplications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sun HotSpot</td>
<td>IBM</td>
</tr>
<tr>
<td></td>
<td>Client</td>
<td>Server</td>
</tr>
<tr>
<td>Sparse, recompute sparseB</td>
<td>1.280</td>
<td>0.928</td>
</tr>
<tr>
<td>Sparse, reuse sparseB</td>
<td>1.005</td>
<td>0.480</td>
</tr>
<tr>
<td>Sparse, RTCG</td>
<td>0.256</td>
<td>0.703</td>
</tr>
</tbody>
</table>

It takes 3.1 ms to generate the second stage code in Sun HotSpot Client VM with gnu.bytecode. Approximately 37.5 KB bytecode is generated; the total space overhead is 130 KB.

Only one matrix multiplication is needed for runtime code generation to pay for itself. Because the generated code is used 100 times, once for each row of \( A \).

Generating JVM code for the second stage

```java
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(varRi); jvmg.emitLoad(varR);
    jvmg.emitArrayLoad(double1D_type);
    jvmg.emitStore(varRi); // Ri = R[i]
    for (int j=0; j<B.cols; j++) {
        // Load Ri
        jvmg.emitLoad(varRi);
        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.emitLoad(varAi);
            jvmg.emitPushDouble(nz.Bkj);
            jvmg.emitLoad(varAi);
            jvmg.emitArrayLoad(Type.double_type);
            load A[i][k] // 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(varRi);
    jvmg.emitAdd('I'); // Ri[i+1] += Ri[i]
    jvmg.emitLoad(varaRows);
    jvmg.emitGotoIfLt(loop); // while (i<aRows);
    jvmg.emitReturn();
}
```

Related work: Two-level source languages instead of bytecode

- Lisp/Scheme backquote (') and comma (,) and eval (MIT Lisp 1978). Polynomial example:
  ```lisp
  (define (polygen cs)
      (if (null? cs)
          '0
          (+ ,0 (car cs) (+ ,0 (* ,x (polygen (cdr cs)))))))
  ```

  For instance, (polygen '((1 1 2 2 3 3)) gives (+ 11 (+ 22 (+ 33 (+ 0 0))))

- Typed two-level Java: DynJava (Owaa et al. 2001); surprisingly poor speed-up. Polynomial example:
  ```java
  '{doubleres=0.0;
   for (int i=cs.length-1;i>=0;i--)
       '{res=res*x+$cs[i];}
  '{returnres;}
  ```

  Types can help avoid binding-time mistakes, and make sure that generated code is well-formed.

- Untyped two-level C: Tick C (Engler et al. 1996); no longer maintained.

- Runtime program specialization in C: The Tempso system (Consel and Noel 1996)

- Untyped two-level OCaml: (bytecode, untyped); Dynamic Caml (Lomov and Moskal 2001).


Conclusions and observations

- The JVM and CLR are interesting, widely available platforms for runtime code generation.
- Bytecode generation plus just-in-time compilation provides for portability and fast generated code.
- Code generation (incl. JIT) is fast: 200,000 instructions/sec for Sun HotSpot Client VM; 100,000 for MS CLR.
- Sun HotSpot Client VM and Microsoft's CLR support runtime code generation well.
- IBM's JVM and Sun HotSpot Server VM are somewhat less suitable.
- For Java, both the gnu.bytecode and BCEL libraries are robust and usable.
- JIT-based implementations use much memory (at least 10 MB), so RTCG is not for embedded systems.
- The performance of JIT-based implementations is somewhat unpredictable, due to adaptive optimizations.

Future work, open questions, student projects, and so on

- Runtime code generation in Moscow ML — e.g. based on a forthcoming backend for CLR.
- Formalize two-level ML for runtime code generation — look at MetaML, MetaCaml, and Dynamic Caml.
- A MetaML with runtime code generation in Moscow ML, using CLR backend?
- Formalize two-level Java — look at DynJava, and Calcagno/Moggi/Sheard's type system.
- Use IBM's Research VM (RVM) to study the effect of JIT-optimizations applied to runtime code generation.
  — Zillions of options, the generated machine code is displayable, new optimizations can be plugged in, …
- Find out whether generated code can be effectively discarded — needed for RTCG in long-running systems.