Runtime code generation

in JVM and .Net CLR *

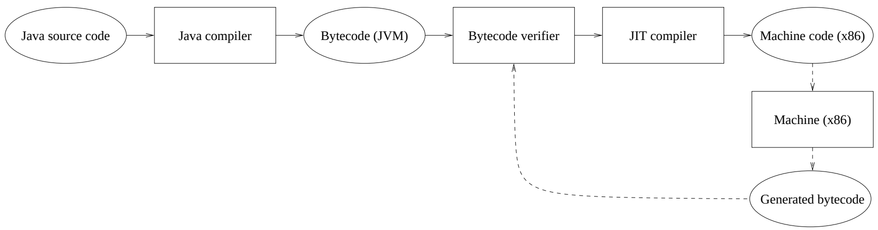
Peter Sestoft (sestoft@dina.kvl.dk)

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:

	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

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 coefficent array $cs[0 \ldots n]$:

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

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

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

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

```
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 coeffient array $cs[\,]$; then (2) for every x, execute the specialized code.

The specialized code for a given polynomial

The constant `cs_i` is the value of $cs[i]$:

```
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`:

```
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

```
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 $cs[\,]$ is long ($\geq$ 20) or many coefficients are zero.

The power example

Computing x^n, that is, x to the n'th power:

```
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;
}
```

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

For $n = 7$, this gives:

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


Generating the function $\texttt{power}_n(x)$ for a fixed n

```
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_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 `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 ILGenerator.

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 = assemblyBuilder.DefineDynamicModule(...);
TypeBuilder typeBuilder     = moduleBuilder.DefineType("MyClass", ...);
MethodBuilder methodBuilder = typeBuilder.DefineMethod("MyMethod", ...);
ILGenerator ilg = methodBuilder.GetILGenerator();
... use ilg to generate the body of method MyClass.MyMethod ...
Type ty = typeBuilder.CreateType();
```

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

```
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 = 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)

```
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];
}
```

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)

```
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;
}
```

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

```
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);
        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]);
  }
}
```


Performance

No computations depend only on the round key `rk[r]`, so one would expect very little speed-up.

But improved instruction scheduling and register allocation (?) do give a speed-up:

	Encryption (Mbit/s)
Somewhat naïve	19.1
Hand-optimized	103.1
Specialized (RTCG)	133.1

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:

```
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:

```
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 B and many different A?

Generating JVM code for the second stage

```
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();
```


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

	100 matrix multiplications				1000 matrix multiplications			
	Sun HotSpot		IBM	MS	Sun HotSpot		IBM	MS
	Client	Server	JVM	CLR	Client	Server	JVM	CLR
Plain	2.002	1.758	1.105	1.410	19.998	16.178	10.595	14.340
Sparse, recompute `sparseB`	1.280	0.928	0.792	1.231	11.860	6.576	7.731	11.806
Sparse, reuse `sparseB`	1.005	0.480	0.677	0.951	10.047	4.526	7.151	9.654
Sparse, RTCG	0.256	0.703	1.296	0.290	1.057	7.450	1.823	0.851

Sun HotSpot 1.4.0 and IBM JIT 1.3.1 for Linux, and MS .Net CLR 1.0 on Windows 2000; 850 MHz Pentium 3.

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.

Related work: Two-level source languages instead of bytecode

- Lisp/Scheme backquote (`‘`) and comma (`,`) and `eval` (MIT Lisp 1978). Polynomial example:

```
(define (polygen cs)
  (if (null? cs)
      ‘0
      ‘(+ ,(car cs) (* x ,(polygen (cdr cs))))))
```

  For instance, `(polygen ’(11 22 33))` gives `(+ 11 (* x (+ 22 (* x (+ 33 (* x 0))))))`

- Typed two-level Java: DynJava (Oiwa et al. 2001); surprisingly poor speed-up. Polynomial example:

```
‘{ double res = 0.0; }
for (int i=cs.length-1; i>=0; i--)
  ‘{ res = res * x + $cs[i]; }
‘{ return res; }
```

- 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 Tempo system (Consel and Noël 1996)
- Untyped two-level OCaml (bytecode, untyped): Dynamic Caml (Lomov and Moskal 2001).
- Multi-level typed ML (machine code): MetaML (Sheard 1998).
- Multi-level typed OCaml (bytecode): MetaOCaml (Calcagno, Taha, Huang, Leroy 2001).
- Code generation tools in C#: Cisternino, Kennedy 2002.

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, MetaOCaml, 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.