# Numeric performance in C, C# and Java

Peter Sestoft (sestoft@itu.dk)

IT University of Copenhagen
Denmark

Version 0.7.1 of 2007-02-28

**Abstract:** We compare the numeric performance of C, C# and Java on three small cases.

## 1 Introduction: Is C# slower than C/C++?

Managed languages such as C# and Java are easier and safer to use than traditional languages such as C or C++ when manipulating dynamic data structures, graphical user interfaces, and so on. Moreover, it is easy to achieve good performance in the managed languages thanks to their built-in automatic memory management.

For numeric computations involving arrays or matrices of floating-point numbers, the situation is less favorable. Compilers for Fortran, C and C++ make serious efforts to optimize inner loops that involve array accesses: register allocation, reduction in strength, common subexpression elimination and so on. By contrast, the just-in-time (JIT) compilers of the C# and Java runtime systems do not spend much time on optimizing inner loops, and this hurts numeric code particularly hard. Moreover, in C# and Java there must be an index check on every array access, and this not only requires execution of extra instructions, but can also lead to branch mispredictions and pipeline stalls on the hardware, further slowing down the computation.

This note explores the reasons for the allegedly poor numeric performance of C# as compared to C. It then goes on to show that a tiny amount of unsafe code can seriously improve the speed of C#.

### 1.1 Case study 1: matrix multiplication

We take matrix multiplication as a prime example of numeric computation. It involves triply nested loops, many array accesses, and floating-point computations, yet the code is so compact that one can study the generated machine code. We find that C performs best, that C# can be made to perform reasonably well, and that Java can perform better than C#. See sections 2 through 5.4.

### 1.2 Case study 2: a division-intensive loop

We also consider a simple loop that performs floating-point division, addition and comparison, but no array accesses. On this rather extreme case we find that C# and Java implementations perform better than C.

### 1.3 Case study 3: polynomial evaluation

Finally we consider repeated evaluation of a polynomial of high degree, on which almost all implementations do equally well, with C and Microsoft C# being equally fast, and Java only slightly slower.

## 2 Matrix multiplication in C

In C, a matrix can be represented by a struct holding the number of rows, the number of columns, and a pointer to a `malloc`'ed block of memory that holds the elements of the matrix as a sequence of doubles:

```
typedef struct {
  int rows, cols;
  double *data;     // (rows * cols) doubles, row-major order
} matrix;
```

If the dimensions of the matrix are known at compile-time, a more static representation of the matrix is possible, but experiments show that for some reason this does not improve speed, quite the contrary.

Given the above struct type, and declarations

```
matrix R, A, B;
```

we can compute the matrix product $R = AB$ in C with this loop:

```
for (r=0; r<rRows; r++) {
  for (c=0; c<rCols; c++) {
    double sum = 0.0;
    for (k=0; k<aCols; k++)
      sum += A.data[r*aCols+k] * B.data[k*bCols+c];
    R.data[r*rCols+c] = sum;
  }
}
```

Note that the programmer must understand the layout of the matrix (here, row-major) and it is his responsibility to get the index computations right.

## 3 Matrix multiplication in C#

### 3.1 Straightforward matrix multiplication in C# (matmult1)

In C# we can represent a matrix as a two-dimensional rectangular array of doubles, using type `double[,]`. Assuming the declaration

```
double[,] R, A, B;
```

we can compute $R = AB$ with this loop:

```
for (int r=0; r<rRows; r++) {
  for (int c=0; c<rCols; c++) {
    double sum = 0.0;
    for (int k=0; k<aCols; k++)
      sum += A[r,k] * B[k,c];
    R[r,c] = sum;
  }
}
```

The variables `rRows`, `rCols` and `aCols` have been initialized from the array dimensions before the loop as follows:

```
int aCols = A.GetLength(1),
    rRows = R.GetLength(0),
    rCols = R.GetLength(1);
```

### 3.2 Unsafe but faster matrix multiplication in C# (matmult2)

The C# language by default requires array bounds checks and disallows pointer arithmetics, but the language provides an escape from these strictures in the form of so-called unsafe code. Hence the C# matrix multiplication code above can be rewritten closer to C style as follows:

```
for (int r=0; r<rRows; r++) {
  for (int c=0; c<rCols; c++) {
    double sum = 0.0;
    unsafe {
      fixed (double* abase = &A[r,0], bbase = &B[0,c]) {
        for (int k=0; k<aCols; k++)
          sum += abase[k] * bbase[k*bCols];
      }
      R[r,c] = sum;
    }
  }
}
```

Inside the `unsafe { ... }` block, one can use C-style pointers and pointer arithmetics. The header of the `fixed (...) { ... }` block obtains pointers `abase` and `bbase` to positions within the `A` and `B` arrays, and all indexing is done off these pointers using C/C++-like notation such as `abase[k]` and `bbase[k*bCols]`. The `fixed` block makes sure that the .NET runtime memory management does not move the arrays `A` and `B` while the block executes. (This risk does not exist in C and C++, where `malloc`'ed blocks stay where they are).

Indexing off a pointer as in `abase[k]` performs no index checks, so this code is riskier but faster than that of the previous section.

Notice that we did not have to change the matrix representation to use unsafe code; we continue to use the `double[,]` representation that is natural in C#.

The `unsafe` keyword may seem scary, but note that *all* code in C and C++ is unsafe in the sense of this keyword. To compile a C# program containing unsafe code, one must pass the `-unsafe` option to the compiler:

```
csc -unsafe MatrixMultiply3.cs
```

## 4 Matrix multiplication in Java

The Java and C# programming languages are managed languages and very similar: same machine model, managed platform, mandatory array bounds checks and so on. There's considerable evidence that Java numeric code can compete with C/C++ numeric code [1, 2, 3].

Some features of Java would seem to make it harder to obtain good performance in Java than in C#:

- Java has only one-dimensional arrays, so a matrix must be represented either as an array of references to arrays of doubles (type `double[][]`) or as a flattened C-style array of doubles (type `double[]`). The former representation can incur a considerable memory access overhead, and the latter representation forces the programmer to explicitly perform index computations.
- Java does not allow unsafe code, so in Java, array bounds checks cannot be circumvented in the way it was done for C# in section 3.2 above.

On the other hand, there is a wider choice of high-performance virtual machines available for Java than for C#. For instance, the "standard" Java virtual machine, namely Hotspot [7] from Sun Microsystems, will aggressively optimize the JIT-generated x86 code if given the `-server` option:

```
java -server MatrixMultiply 80 80 80
```

The Sun Hotspot Java virtual machine defaults to `-client`, which favors quick start-up over fast generated code, as preferable for most interactive programs.

Also, IBM's Java virtual machine [8] appears to perform considerable optimizations when generating machine code from the bytecode. There are further high-performance Java virtual machines, such as BEA's jrockit [9], but we have not tested them.

As mentioned, the natural Java representation of a two-dimensional matrix is an array of references to arrays (rows) of doubles, that is, Java type `double[][]`. Assuming the declaration

```
double[][] R, A, B;
```

The corresponding matrix multiplication code looks like this:

```
for (int r=0; r<rRows; r++) {
  double[] Ar = A[r], Rr = R[r];
  for (int c=0; c<rCols; c++) {
    double sum = 0.0;
    for (int k=0; k<aCols; k++)
      sum += Ar[k]*B[k][c];
    Rr[c] = sum;
  }
}
```

Here we have made a small optimization, in that references `Ar` and `Rr` to the arrays `A[r]` and `R[r]`, which represent rows of `A` and `R`, are obtained at the beginning of the outer loop.

The array-of-arrays representation seems to give the fastest matrix multiplication in Java. Replacing it with a one-dimensional array (as in C) makes matrix multiplication 1.3 times slower.

# 5 Compilation of matrix multiplication code

This section presents the bytecode and machine code obtained by compiling the matrix multiplication source codes shown in the previous section, and discusses the speed and deficiencies of this code.

## 5.1 Compilation of the C matrix multiplication code

Recall the inner loop

```
for (k=0; k<aCols; k++)
  sum += A.data[r*aCols+k] * B.data[k*bCols+c];
```

of the C matrix multiplication code in section 2. The x86 machine code generated for this inner loop by the gcc compiler with full optimization (`gcc -O3`) is quite remarkably brief:

```
      <loop header not shown>
.L45:
      fldl    (%ecx)          // load B.data[k*bCols+c]
      addl    %ebx, %ecx      // add 8*bCols to B.data index
      fmull   (%edx)          // multiply with A.data[r*aCols+k]
      addl    $8, %edx        // add 8 to A.data index
      decl    %eax            // decrement loop counter
      faddp   %st, %st(1)     // sum += ...
      jne     .L45            // jump to L45 if loop counter non-zero
```

This loop takes time 2.8 ns per iteration on a 1600 MHz Pentium M CPU, so it also exploits the hardware parallelism and the data buses very well. See also section 9 below.

## 5.2 Compilation of the safe C# code

C# source code, like Java source code, gets compiled in two stages:

- First the C# code is compiled to stack-oriented bytecode in the .NET Common Intermediate Language (CIL), using the Microsoft `csc` compiler [6], possibly through Visual Studio, or using the Mono `mcs` or `gmcs` compiler [10]. The result is a so-called Portable Executable file, named MatrixMultiply.exe or similar, which consists of a stub to invoke the .NET Common Language Runtime, the bytecode, and some metadata.
- Second, when the compiled program is about to be executed, the just-in-time compiler of the Common Language Runtime will compile the stack-oriented bytecode to register-oriented machine code for the real hardware (typically some version of the x86 architecture). Finally the generated machine code is executed. The just-in-time compilation process can be fairly complicated and unpredictable, with profiling-based dynamic optimization and so on.

Recall the inner loop of the straightforward C# matrix multiplication (matmult1) in section 3.1:

```
for (int k=0; k<aCols; k++)
  sum += A[r,k] * B[k,c];
```

The corresponding CIL bytecode generated by the Microsoft C# compiler `csc -o` looks like this:

```
<loop header not shown>
IL_005a:  ldloc.s    V_8            // load sum
IL_005c:  ldarg.1                   // load A
IL_005d:  ldloc.s    V_6            // load r
IL_005f:  ldloc.s    V_9            // load k
IL_0061:  call float64[,]::Get(,)   // load A[r,k]
IL_0066:  ldarg.2                   // load B
IL_0067:  ldloc.s    V_9            // load k
IL_0069:  ldloc.s    V_7            // load c
IL_006b:  call float64[,]::Get(,)   // load B[k,c]
IL_0070:  mul                       // A[r,k] * B[k,c]
IL_0071:  add                       // sum + ...
IL_0072:  stloc.s    V_8            // sum = ...
IL_0074:  ldloc.s    V_9            // load k
IL_0076:  ldc.i4.1                  // load 1
IL_0077:  add                       // k+1
IL_0078:  stloc.s    V_9            // k = k+1
IL_007a:  ldloc.s    V_9            // load k
IL_007c:  ldloc.1                   // load aCols
IL_007d:  blt.s      IL_005a        // jump if k<aCols
```

As can be seen, this is straightforward stack-oriented bytecode which hides the details of array bounds checks and array address calculations inside the `float64[,]::Get(,)` method calls.

One can obtain the x86 machine code generated by the Mono runtime's just-in-time compiler by invoking it as `mono -v -v`. The resulting x86 machine code is rather cumbersome (and slow) because of the array address calculations and the array bounds checks. These checks and calculations are explicit in the x86 code below; the `Get(,)` method calls in the bytecode have been inlined:

```
 <loop header not shown>
 f8:        fldl   0xffffffd4(%ebp)         // load sum
 fb:        fstpl  0xffffffcc(%ebp)
 fe:        mov    0xc(%ebp),%eax           // array bounds check
101:        mov    %eax,0xffffffb8(%ebp)    // array bounds check
104:        mov    0x8(%eax),%eax           // array bounds check
107:        mov    0x4(%eax),%edx           // array bounds check
10a:        mov    %esi,%ecx                // array bounds check
10c:        sub    %edx,%ecx                // array bounds check
10e:        mov    (%eax),%edx              // array bounds check
110:        cmp    %ecx,%edx                // array bounds check
112:        jbe    22e                      // throw exception
118-12a:    // array bounds check (not shown)
130:        imul   %ecx,%eax
133:        mov    0xffffffb8(%ebp),%ecx
136:        add    %edx,%eax
138:        imul   $0x8,%eax,%eax
13b:        add    %ecx,%eax
13d:        add    $0x10,%eax
142:        fldl   (%eax)                   // load A[r][k]
144:        fstpl  0xffffffc4(%ebp)
147:        fldl   0xffffffcc(%ebp)
14a:        fldl   0xffffffc4(%ebp)
14d-161:    // array bounds check (not shown)
167-179:    // array bounds check (not shown)
17f:        imul   %ecx,%eax
182:        mov    0xffffffc0(%ebp),%ecx
185:        add    %edx,%eax
187:        imul   $0x8,%eax,%eax
18a:        add    %ecx,%eax
18c:        add    $0x10,%eax
191:        fldl   (%eax)                   // load B[k][c]
193:        fmulp  %st,%st(1)               // multiply
195:        faddp  %st,%st(1)               // add sum
197:        fstpl  0xffffffd4(%ebp)         // sum = ...
19a:        inc    %edi                     // increment k
<BB>:12
19b:        cmp    0xffffffec(%ebp),%edi
19e:        jl     f8                       // jump if k<aCols
```

Registers: `%esi` holds `r`, `%ebx` holds `c`, `%edi` holds `k`. For brevity, some repetitive sections of code are not shown.

According to experiments, this x86 code is approximately 9 times slower than the code generated from C source by `gcc -O3` and shown in section 5.1. The x86 code generated by Microsoft's just-in-time compiler is slower than the `gcc` code only by a factor of 4, and presumably also looks neater.

### 5.3 Compilation of the unsafe C# code

Now let us consider the unsafe (matmult2) version of the C# matrix multiplication code from section 3.2. The inner loop looks like this:

```
fixed (double* abase = &A[r,0], bbase = &B[0,c]) {
  for (int k=0; k<aCols; k++)
    sum += abase[k] * bbase[k*bCols];
}
```

The CIL bytecode generated by Microsoft's C# compiler looks like this:

```
<loop header not shown>
IL_0079:  ldloc.s    V_8            // load sum
IL_007b:  ldloc.s    V_9            // load abase
IL_007d:  conv.i
IL_007e:  ldloc.s    V_11           // load k
IL_0080:  conv.i
IL_0081:  ldc.i4.8                  // load 8
IL_0082:  mul                       // 8*k
IL_0083:  add                       // abase+8*k
IL_0084:  ldind.r8                  // load abase[k]
IL_0085:  ldloc.s    V_10           // load bbase
IL_0087:  conv.i
IL_0088:  ldloc.s    V_11           // load k
IL_008a:  ldloc.3                   // load bCols
IL_008b:  mul                       // k*bCols
IL_008c:  conv.i
IL_008d:  ldc.i4.8                  // load 8
IL_008e:  mul                       // 8*k*bCols
IL_008f:  add                       // bbase+8*k*bCols
IL_0090:  ldind.r8                  // load bbase[k*bCols]
IL_0091:  mul                       // multiply
IL_0092:  add                       // add sum
IL_0093:  stloc.s    V_8            // sum = ...
IL_0095:  ldloc.s    V_11           // load k
IL_0097:  ldc.i4.1                  // load 1
IL_0098:  add                       // k+1
IL_0099:  stloc.s    V_11           // k = ...
IL_009b:  ldloc.s    V_11           // load k
IL_009d:  ldloc.1                   // load aCols
IL_009e:  blt.s      IL_0079        // jump if k<aCols
```

At first sight this appears even longer and more cumbersome than the matmult1 bytecode sequence in section 5.2, but note that it does not involve any calls to the `float64[,]::Get(,)` methods, and hence does not contain any hidden costs.

The corresponding x86 machine code generated by Mono now is much shorter:

```
<loop header not shown>
1b8:        fldl   0xffffffcc(%ebp)         // load sum
1bb:        mov    %ebx,%eax                // load k
1bd:        mov    %eax,%ecx
1bf:        shl    $0x3,%ecx                // 8*k
1c2:        mov    %edi,%eax                //
1c4:        add    %ecx,%eax                // abase+8*k
1c6:        fldl   (%eax)                   // load abase[k]
1c8:        mov    %ebx,%eax                // load k
1ca:        imul   0xfffffffe4(%ebp),%eax   // k*bCols
1ce:        mov    %eax,%ecx
1d0:        shl    $0x3,%ecx                // 8*k*bCols
1d3:        mov    %esi,%eax                // load bbase
1d5:        add    %ecx,%eax                // base+8*k*bCols
1d7:        fldl   (%eax)                   // load bbase[k*bCols]
1d9:        fmulp  %st,%st(1)               // multiply
1db:        faddp  %st,%st(1)               // add sum
```

```
 1dd:        fstpl  0xffffffcc(%ebp)         // sum = ...
 1e0:        inc    %ebx                     // increment k
<BB>:12
 1e1:        cmp    0xffffffec(%ebp),%ebx    //
 1e4:        jl     1b8 <MyTest_Multiply+0x1b8>// jump if k<aCols
```

Registers: `%ebx` holds `k`, `%ebi` holds `abase`, `%esi` holds `bbase`.

Clearly this unsafe code is far shorter than the x86 code in section 5.2 that resulted from safe bytecode. One iteration of this loop takes 15.5 ns on a 1600 MHz Pentium M.

However, one iteration of the corresponding x86 code generated by Microsoft's just-in-time compiler takes only 4.0 ns, so presumably address multiplications have been replaced by additions (reduction in strength), frequently used variables such as `bCols` and `aCols` are kept in registers rather than in memory, and so on.

Microsoft's Visual Studio development environment does allow one to inspect the x86 code generated by the just-in-time compiler, but only when debugging a C# program: Set a breakpoint in the method whose x86 you want to see, choose `Debug | Start debugging`, and when the process stops, choose `Debug | Windows | Disassembly`. Unfortunately, the x86 code shown during debugging clearly contains extraneous and wasteful instructions, such as those at addresses 182-185 and 18e-191:

```
<loop header not shown>
0000017f  mov   eax,dword ptr [ebp-34h]      // load abase
00000182  mov   dword ptr [ebp-54h],eax      // move it to RAM ...
00000185  mov   eax,dword ptr [ebp-54h]      // ... and move it back
00000188  fld   qword ptr [eax+ebx*8]        // load abase[k]
0000018b  mov   eax,dword ptr [ebp-38h]      // load bbase
0000018e  mov   dword ptr [ebp-58h],eax      // move it to RAM ...
00000191  mov   eax,dword ptr [ebp-58h]      // ... and move it back
00000194  mov   edx,dword ptr [ebp-1Ch]      // load bCols
00000197  imul  edx,ebx                      // k*bCols
0000019a  fmul  qword ptr [eax+edx*8]        // multiply by bbase[]
0000019d  fadd  qword ptr [ebp-30h]          // add sum
000001a0  fstp  qword ptr [ebp-30h]          // sum = ...
000001a3  inc   ebx                          // increment k
000001a4  cmp   ebx,dword ptr [ebp-14h]      // compare k<aCols
000001a7  jl    0000017F                     // jump if k<aCols
```

In fact, the x86 code shown takes twice as long time, 8 ns per loop iteration, as non-debugging code. Hence the x86 code obtained from Visual Studio during debugging does not give a good indication of the code quality that is actually achievable. To avoid truly bad code, make sure to tick off the `Optimize` checkbox in the `Project | Properties | Build` form in Visual Studio.

The machine code generated by Microsoft's just-in-time compiler can be inspected also using the debugger `cordbg` from the command line, setting a breakpoint in a method, and disassembling it, like this

```
C:\> cordbg MatrixMultiply3.exe 50 50 50
(cordbg) b MatrixMultiply::Multiply  // breakpoint on method Multiply
(cordbg) g                           // execute to breakpoint
(cordbg) dis 1000                    // disassemble 1000 instructions
(cordbg)
```

But again it seems that the C# program must be compiled using the `csc -debug` flag for this to work, and the x86 machine code does not seem optimal even when `csc -o` has been specified.

### 5.4 Compilation of the Java matrix multiplication code

The bytecode resulting from compiling the Java code in section 4 with the Sun Java compiler `javac` is fairly similar to the CIL bytecode code shown in section 5.2.

Remarkably, the straightforward Java implementation, which uses no unsafe code and a seemingly cumbersome array representation, performs better than the unsafe C# code when executed with the IBM Java virtual machine [8]: Each iteration of the inner loop takes only 3.8 ns.

Execution with the Sun Hotspot client virtual machine is slower, but the `-server` option reduces the execution time to within a factor 1.2 of IBM's, far better than the straightforward C# implementation, and still using only safe code. This seems to demonstrate that it is handy to be able to choose between different optimization profiles, such as `-client` and `-server`.

In Sun's so-called DEBUG or fastdebug versions `java_g` of beta versions of the Hotspot Java runtime environment, the non-standard option `-XX:+PrintOptoAssembly` is reported to show the x86 code generated by the just-in-time compiler. We have not investigated this.

## 6 Controlling the runtime and the just-in-time compiler

I know of no publicly available options or flags to control the just-in-time optimizations performed by Microsoft's .NET Common Language Runtime, but surely options similar to Sun's `-client` and `-server` must exist internally. I know that there is (or was) a Microsoft-internal tool called `jitmgr` for configuring the .NET runtime and just-in-time compiler, but it does not appear to be publicly available. Presumably many people would just use it to shoot themselves in the foot.

Note that the so-called server build (mscorsvr.dll) of the Microsoft .NET runtime differs from the workstation build (mscorwks.dll) primarily in using a concurrent garbage collector. According to MSDN, the workstation build will always be used on uniprocessor machines, even if the server build is explicitly requested.

The Mono runtime accepts a range of JIT optimization flags, such as

```
mono --optimize=loop MatrixMultiply 80 80 80
```

but at the time of writing (Mono version 1.2.3, February 2007), the effect of these flags seems modest and somewhat erratic.

## 7 Case study 2: A division-intensive loop

Consider for a given $M$ the problem of finding the least integer $n$ such that

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq M$$

In C, Java and C# the problem can be solved by the following program fragment:

```
double sum = 0.0;
int n = 0;
while (sum < M) {
  n++;
  sum += 1.0/n;
}
```

For $M = 20$ the answer is $n = 272\ 400\ 600$ and the loop performs that many iterations. Each iteration involves a floating-point comparison, a floating-point division and a floating-point addition, as well as an integer increment.

The computation time is dominated by the double-precision floating-point division operation FDIV, which has a throughput of 32 cycles/instruction on the Pentium M [5]. Since the loop condition depends on the division, this gives a lower bound of 20 ns per loop iteration on the 1600 MHz machine we are using. Indeed all implementations take between 20.7 ns and 26.0 ns per iteration. Interestingly, IBM's Java virtual machine is the fastest and the C implementation (`gcc -O3`) is the slowest of these.

The x86 machine code generated by the Mono just-in-time compiler from the C# is this:

```
<loop header not shown>
78:  inc    %ebx                    // increment n
79:  fldl   0xfffffff0(%ebp)        // load sum
7c:  fld1                           // load 1.0
7e:  push   %ebx                    // push n on hardware stack
7f:  fildl  (%esp)                  // load n on fp stack
82:  add    $0x4,%esp               // pop hardware stack
85:  fdivrp %st,%st(1)              // divide 1.0/n
87:  faddp  %st,%st(1)              // add sum
89:  fstpl  0xfffffff0(%ebp)        // sum = ...
8c:  fldl   0xfffffff0(%ebp)        // load sum
8f:  fldl   0xfffffff8(%ebp)        // load M
92:  fcomip %st(1),%st              // compare sum<M
94:  fstp   %st(0)
96:  ja     78                      // jump if sum<M
```

## 8 Polynomial evaluation

A polynomial $c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$ can be evaluated efficiently and accurately using Horner's rule:

$$c_0 + c_1x + c_2x^2 + \cdots + c_nx^n = c_0 + x \cdot (c_1 + x \cdot (\ldots + x \cdot (c_n + x \cdot 0)\ldots))$$

Polynomial evaluation using Horner's rule can be implemented in C, Java and C# like this:

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

where the coefficient array `cs` has length $n + 1$ and coefficient $c_i$ is in element `cs[`$n - i$`]`.

The x86 code generated for the above polynomial evaluation loop by `gcc -O3` from C is this:

```
        <loop header not shown>
.L21:
        fmul    %st(2), %st             // multiply by x
        faddl   (%esi,%eax,8)           // add cs[i]
        incl    %eax                    // increment i
        cmpl    %ebx, %eax              // compare i<order
        jl      .L21                    // jump if i<order
```

Note that the entire computation is done with `res` on the floating-point stack; not once during the loop is anything written to memory. The array accesses happen in the `faddl` instruction.

All implementations fare almost equally well on this problem, with C# on Microsoft's .NET being the fastest at 5.1 ns per loop iteration, and the `gcc -O3` compiled C code only slightly slower at 5.3 ns per loop iteration. Each iteration performs a floating-point addition and a multiplication, but here the

multiplication uses the result of the preceding addition (via a loop-carried dependence), which may be the reason this is so much slower than the matrix multiplication loop in section 5.1.

The reason for the Microsoft implementation's excellent performance may be that it can avoid the array bounds check in `cs[i]`. The just-in-time compiler can recognize bytecode generated from loops of exactly this form:

```
for (int i=0; i<cs.Length; i++)
  ... cs[i] ...
```

and will not generate array bounds checks for the `cs[i]` array accesses [11]. Apparently this optimization is rather fragile; small deviations from the above code pattern will prevent the just-in-time compiler from eliminating the array bounds check. Also, experiments confirm that this optimization is useless in the safe matrix multiplication loop (section 3.1), where at least two of the four index expressions appear not to be bounded by the relevant array length (although in reality they are, of course).

# 9 Experiments

## 9.1 Matrix multiplication performance

This table shows the CPU time (in microseconds) per matrix multiplication, for multiplying two 50x50 matrices and for multiplying two 80x80 matrices:

| Hardware | Pentium M 1.6 | | Pentium 4 2.8 | |
|---|---|---|---|---|
| Matrix size | 50x50 | 80x80 | 50x50 | 80x80 |
| C (`gcc -O3`) | 340 | 1430 | 275 | 1130 |
| C# matmult1 Microsoft | 1320 | 5500 | 1120 | 4960 |
| C# matmult1 Microsoft, ngen | 1320 | 5550 | 1120 | 4960 |
| C# matmult1 Mono | 3010 | 12350 | 2675 | 11880 |
| C# matmult2 Microsoft | 500 | 2020 | 490 | 2055 |
| C# matmult2 Microsoft, ngen | 500 | 1920 | 480 | 2100 |
| C# matmult2 Mono | 1950 | 7900 | 1430 | 5840 |
| Java, Sun Hotspot -client | 840 | 3380 | 570 | 2335 |
| Java, Sun Hotspot -server | 618 | 2360 | 370 | 1625 |
| Java, IBM JVM | 465 | 1950 | 345 | 1705 |

We see that the best C# results are a factor of 1.35 slower than the best C results, using unsafe features of C#. The best Java results, using IBM's JVM, are only a factor 1.37 slower than the best C results, which is impressive considering that no unsafe code is used. It seems that the Microsoft C# speed can sometimes be improved by ahead-of-time compilation of the bytecode to x86 machine code, using the "native image generator" tool `ngen`, like this:

```
C:\> ngen install MatrixMultiply3.exe
C:\> MatrixMultiply3 80 80 80
C:\> ngen uninstall MatrixMultiply3.exe
```

However, if the inner loop contains calls to other assemblies, such as the `Get` methods used for non-inlined array accesses, then ngen'ed programs actually become slower.

Depending on circumstances, the resulting C# performance may be entirely acceptable, given that the unsafe code can be isolated to very small fragments of the code base, and the advantages of safe code and dynamic memory management can be exploited everywhere else. Also, in 2010 a standard workstation may have 16 or 32 CPUs, and then it will probably be more important to exploit parallel computation than to achieve raw single-processor speed.

### 9.2 Division-intensive loop performance

For the simple division-intensive loop shown in section 7 the execution times are as follows, in nanoseconds per iteration of the loop:

| | Pentium M 1.6 | Pentium 4 2.8 |
|---|---|---|
| C (`gcc -O3`) | 26.0 | 16.0 |
| C# Microsoft | 21.1 | 14.4 |
| C# Microsoft, ngen | 20.9 | 14.3 |
| C# Mono | 20.9 | 14.4 |
| Java, Sun Hotspot -client | 21.3 | 14.0 |
| Java, Sun Hotspot -server | 21.1 | 14.0 |
| Java, IBM JVM | 20.7 | 14.0 |

Again the IBM Java virtual machine performs very well, near the theoretical minimum of 20 ns.

### 9.3 Polynomial evaluation performance

The execution times for evaluation of a polynomial of order 1000 (microseconds per polynomial evaluation), implemented as in section 8 are as follows:

| | Pentium M 1.6 | Pentium 4 2.8 |
|---|---|---|
| C (`gcc -O3`) | 5.3 | 4.7 |
| C# Microsoft | 5.1 | |
| C# Microsoft, ngen | 5.1 | |
| C# Mono | 8.4 | 13.0 |
| Java, Sun Hotspot -client | 5.9 | |
| Java, Sun Hotspot -server | 5.1 | |
| Java, IBM JVM | 5.2 | |

The C and Microsoft C# performance must be considered identical, with Sun's Hotspot `-server` and IBM's Java virtual machine close behind. The Mono C# implementation is a factor of 1.5 slower than the best performance in this case.

### 9.4 Details of the experimental platform

- Main hardware platform: Intel Pentium M at 1600 MHz, 1024 KB L2 cache, 1 GB RAM.
- Alternative hardware platform: Intel Pentium 4 at 2800 MHz, 512 KB L2 cache, 1.5 GB RAM.
- Operating system: Debian Linux, kernel 2.4.
- C compiler: gcc 3.3.5 optimization level -O3
- Microsoft C# compiler and runtime: MS .NET 2.0.50727 running under Windows 2000 under VmWare under Linux; compile options: `-o -unsafe`.
- Mono C# compiler and runtime: `gmcs` and `mono` version 1.2.3 for Linux.
- Java compiler and runtime Sun Hotspot 1.6.0-b105 for Linux x86-32.
- Java runtime IBM J9 VM J2RE 1.5.0 for Linux x86-32 j9vmxi3223-20070201.

## 10 Conclusion

The experiments show that there is no obvious relation between the execution speeds of different software platforms, even for the very simple programs studied here: the C, C# and Java platforms are variously fastest and slowest.

Moreover, the Intel Pentium M (which is the basis of future Intel processors) and Pentium 4 hardware platforms are so different that even when the Pentium 4 clock rate is 75 percent higher, some programs run more slowly on that platform, while others become much faster.

Some points that merit special attention:

- Given Java's cumbersome array representation and the absence of unsafe code, it is remarkable how well the Sun Hotspot `-server` and IBM Java virtual machines perform.
- Microsoft's C#/.NET runtime performs very well, but there is room for much improvement in the safe code for matrix multiplication.
- Microsoft's `ngen` tool could do a much better job of (optionally) optimizing numeric code.
- The Mono C#/.NET runtime now is very reliable, but its general performance and the effect of optimization flags is rather erratic.

## References


[1] J.P. Lewis and Ulrich Neumann: Performance of Java versus C++. University of Southern California 2003.
`http://www.idiom.com/~zilla/Computer/javaCbenchmark.html`

[2] National Institute of Standards and Technology, USA: JavaNumerics.
`http://math.nist.gov/javanumerics/`

[3] CERN and Lawrence Berkeley Labs, USA: COLT Project, Open Source Libraries for High Performance Scientific and Technical Computing in Java.
`http://dsd.lbl.gov/~hoschek/colt/`

[4] P. Sestoft: Java performance. Reducing time and space consumption. KVL 2005.
`http://www.dina.kvl.dk/~sestoft/papers/performance.pdf`

[5] Intel 64 and IA-32 Architectures Optimization Reference Manual. November 2006.
`http://www.intel.com/design/processor/manuals/248966.pdf`

[6] Microsoft Developer Network: .NET Framework Developer Center.
`http://msdn.microsoft.com/netframework/`

[7] The Sun Hotspot Java virtual machine is found at `http://java.sun.com`

[8] The IBM Java virtual machine is found at
`http://www-128.ibm.com/developerworks/java/jdk/`

[9] The BEA jrockit Java virtual machine is found at
`http://www.bea.com/content/products/jrockit/`

[10] The Mono implementation of C# and .NET is found at `http://www.mono-project.com/`

[11] Gregor Noriskin: Writing high-performance managed applications. A primer. Microsoft, June 2003. At `http://msdn2.microsoft.com/en-us/library/ms973858.aspx`