Let's talk about microbenchmarking

Andrey Akinshin, JetBrains

DotNext 2016 Helsinki

BenchmarkDotNet

dotnet / **BenchmarkDotNet**

Watch 117 | Star 1,480 | Fork 144

753 commits | **3** branches | **23** releases | **25** contributors | MIT

- Standard benchmarking routine: generating an isolated project per each benchmark method; auto-selection of iteration amount; warmup; overhead evaluation; statistics calculation; and so on.
- Supported runtimes: Full .NET Framework, .NET Core (RTM), Mono
- Supported languages: C#, F#, and Visual Basic
- Supported OS: Windows, Linux, MacOS
- Easy way to compare different environments (`x86` vs `x64` , `LegacyJit` vs `RyuJit` , and so on; see: Jobs)
- Reports: markdown, csv, html, plain text, png plots.
- Advanced features: Baseline, Params
- Powerful diagnostics based on ETW events (see BenchmarkDotNet.Diagnostics.Windows)

This project is supported by the .NET Foundation.

Today's environment

- BenchmarkDotNet v0.10.1
- Haswell Core i7-4702MQ CPU 2.20GHz, Windows 10
- .NET Framework 4.6.2 + clrjit/compatjit-v4.6.1586.0
- Source code: https://git.io/v1RRX

Today's environment

- BenchmarkDotNet v0.10.1
- Haswell Core i7-4702MQ CPU 2.20GHz, Windows 10
- .NET Framework 4.6.2 + clrjit/compatjit-v4.6.1586.0
- Source code: https://git.io/v1RRX

Other environments:

C# compiler	old csc / Roslyn
CLR	CLR2 / CLR4 / CoreCLR / Mono
OS	Windows / Linux / MacOS / FreeBSD
JIT	LegacyJIT-x86 / LegacyJIT-x64 / RyuJIT-x64
GC	MS (different CLRs) / Mono (Boehm/Sgen)
Toolchain	JIT / NGen / .NET Native
Hardware	∞ different configurations
...	...

And don't forget about multiple versions

A bad benchmark

```
// Resolution (Stopwatch) = 466 ns
// Latency    (Stopwatch) = 18 ns
var sw = Stopwatch.StartNew();
Foo(); // 100 ns
sw.Stop();
WriteLine(sw.ElapsedMilliseconds);
```

A bad benchmark

```
// Resolution (Stopwatch) = 466 ns
// Latency    (Stopwatch) = 18 ns
var sw = Stopwatch.StartNew();
Foo(); // 100 ns
sw.Stop();
WriteLine(sw.ElapsedMilliseconds);
```

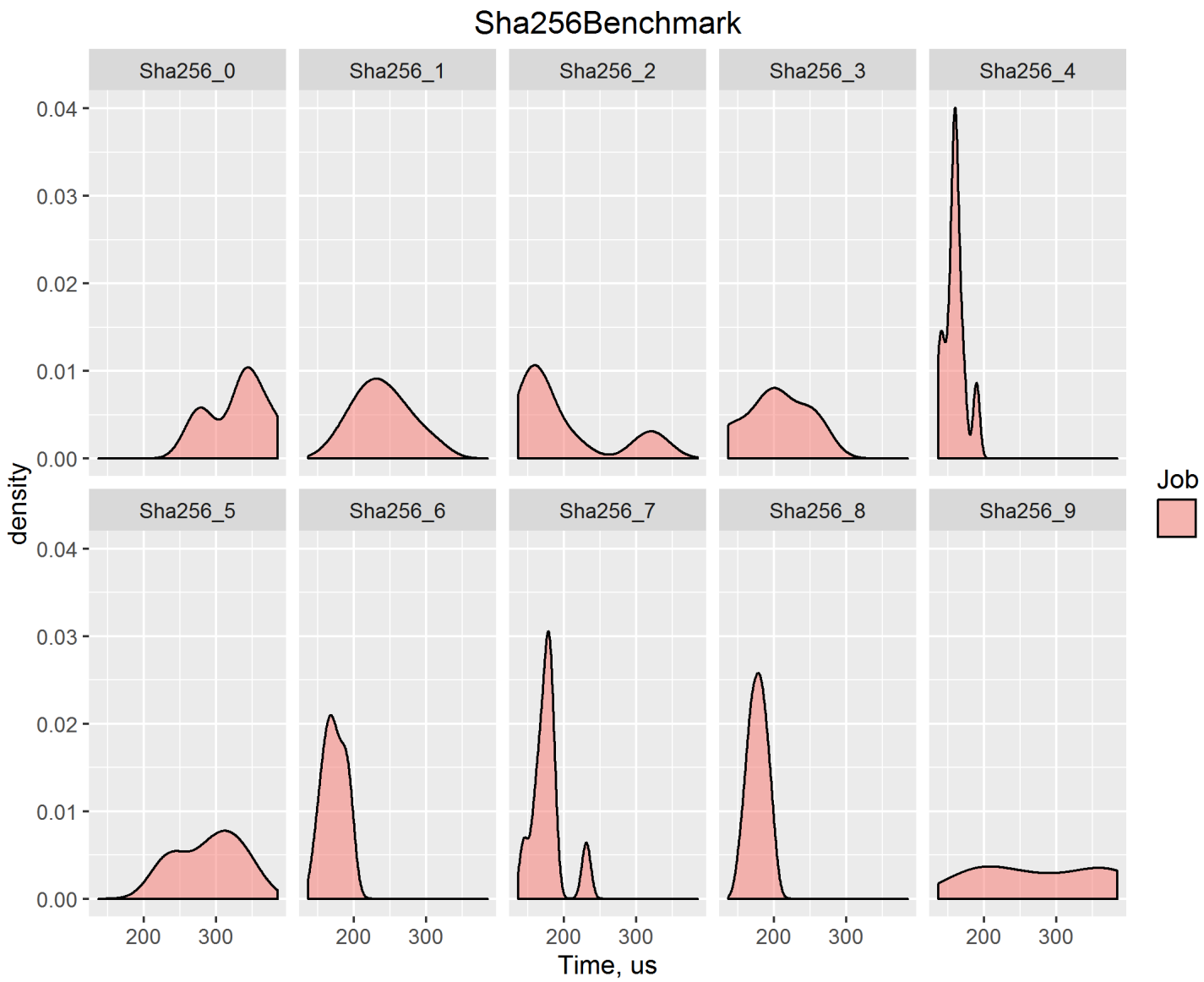
A better benchmark

```
var sw = Stopwatch.StartNew();
for (int i = 0; i < N; i++) // (N * 100 + eps) ns
    Foo(); // 100 ns
sw.Stop();
var total = sw.ElapsedTicks / Stopwatch.Frequency;
WriteLine(total / N);
```

Several launches

```
Run 01 : 529.8674 ns/op
Run 02 : 532.7541 ns/op
Run 03 : 558.7448 ns/op
Run 04 : 555.6647 ns/op
Run 05 : 539.6401 ns/op
Run 06 : 539.3494 ns/op
Run 07 : 564.3222 ns/op
Run 08 : 551.9544 ns/op
Run 09 : 550.1608 ns/op
Run 10 : 533.0634 ns/op
```

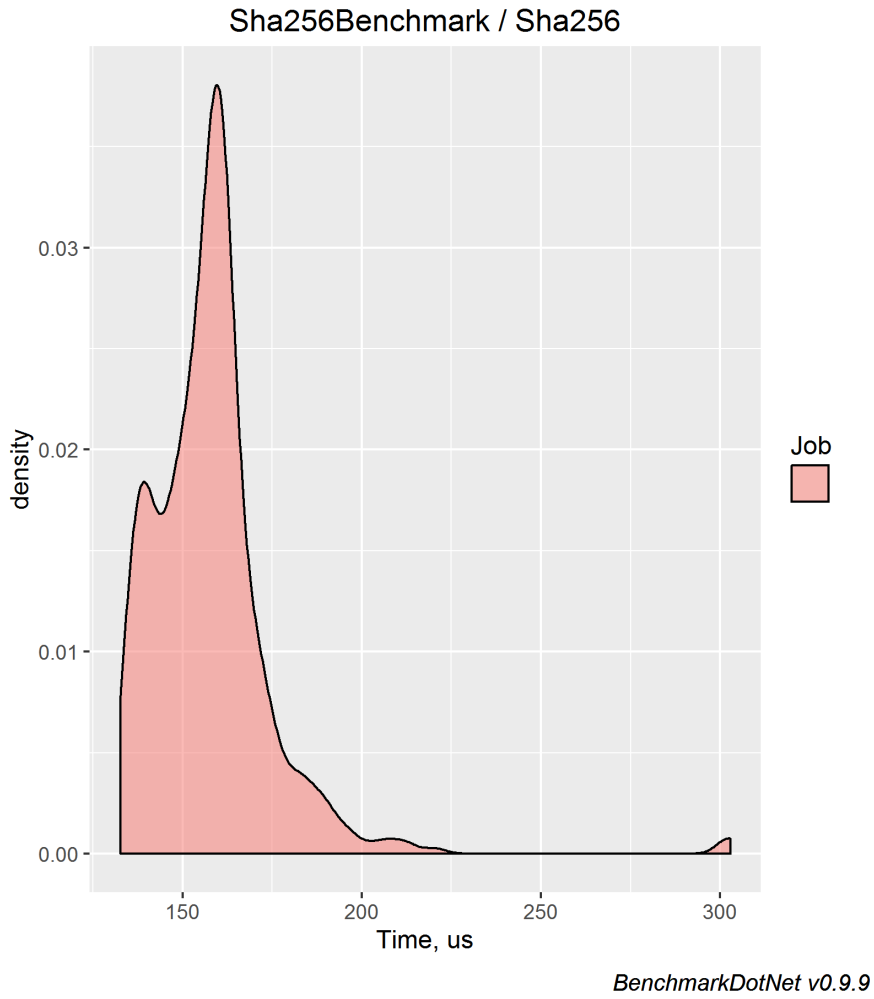
Several launches



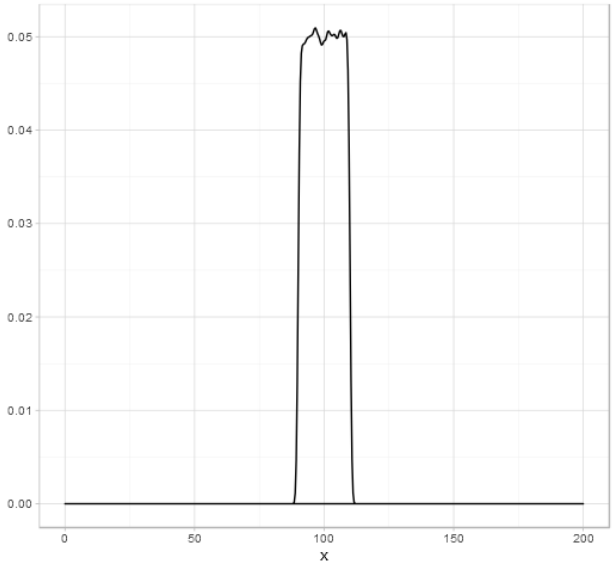
BenchmarkDotNet v0.9.9

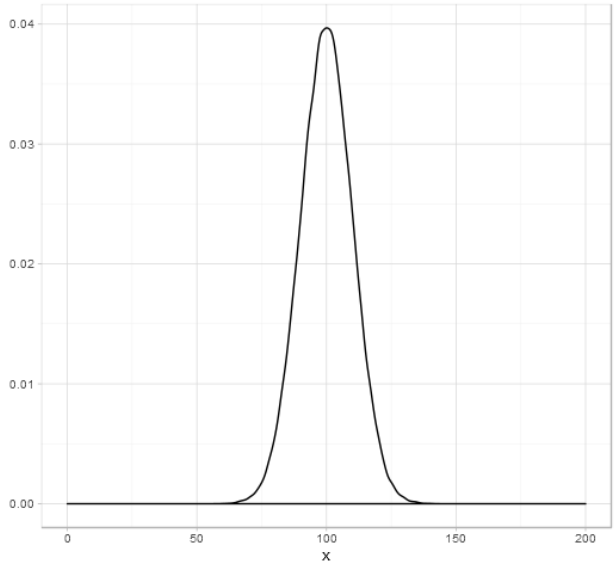
A simple case

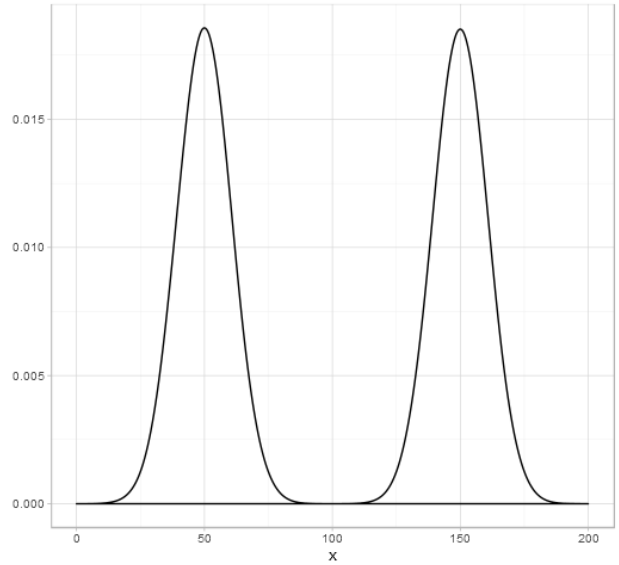
Central limit theorem to the rescue!

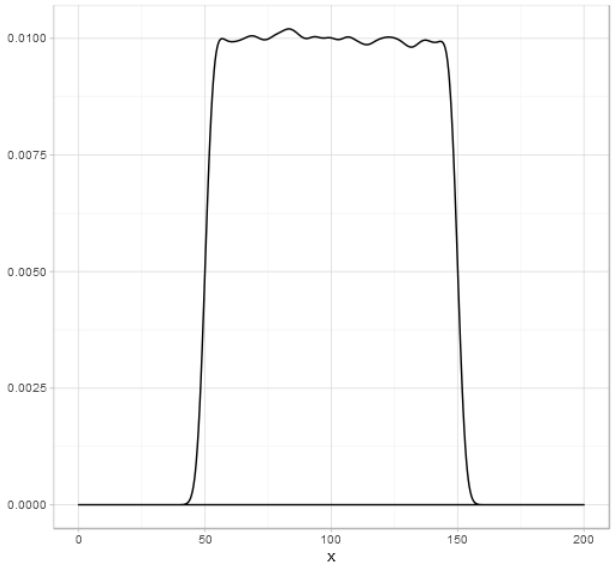


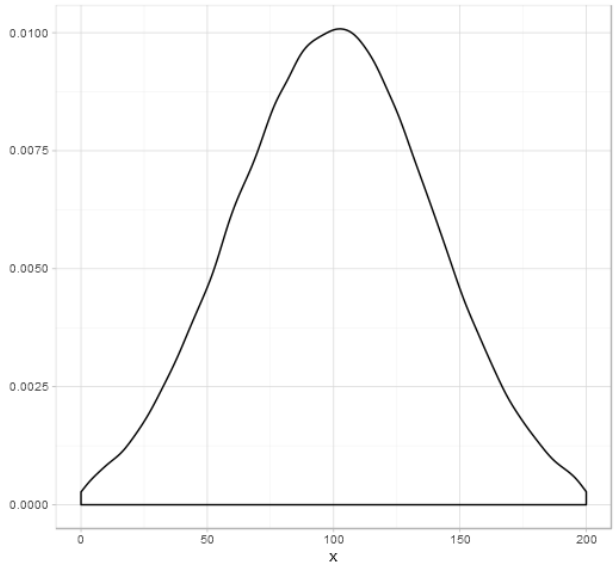
Complicated cases

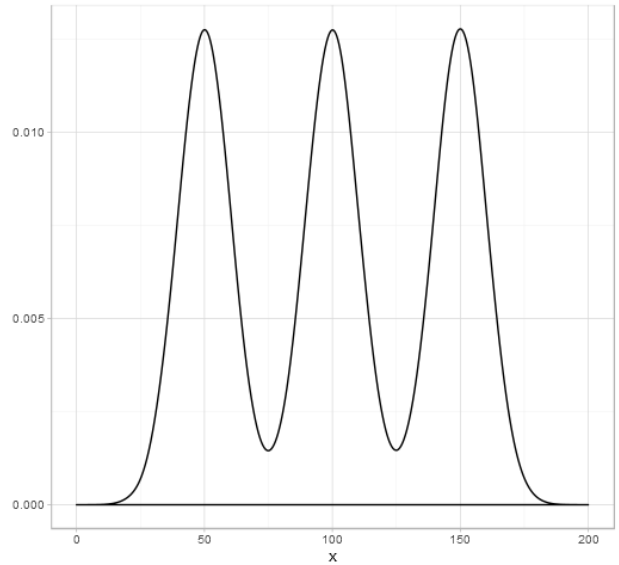












Latencies

Event	Latency
1 CPU cycle	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	2.8 ns
Level 3 cache access	12.9 ns
Main memory access	120 ns
Solid-state disk I/O	50-150 μs
Rotational disk I/O	1-10 ms
Internet: SF to NYC	40 ms
Internet: SF to UK	81 ms
Internet: SF to Australia	183 ms
OS virtualization reboot	4 sec
Hardware virtualization reboot	40 sec
Physical system reboot	5 min

Sum of elements

```
const int N = 1024;
int[,] a = new int[N, N];
```

```
[Benchmark]
public double SumIJ()
{
  var sum = 0;
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      sum += a[i, j];
  return sum;
}
```

```
[Benchmark]
public double SumJI()
{
  var sum = 0;
  for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
      sum += a[i, j];
  return sum;
}
```

Sum of elements

```
const int N = 1024;
int[,] a = new int[N, N];
```

```
[Benchmark]
public double SumIJ()
{
  var sum = 0;
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      sum += a[i, j];
  return sum;
}
```

```
[Benchmark]
public double SumJI()
{
  var sum = 0;
  for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
      sum += a[i, j];
  return sum;
}
```

CPU cache effect:

	SumIJ	SumJI
LegacyJIT-x86	≈1.3ms	≈4.0ms

Cache-sensitive benchmarks

Let's run a benchmark several times:

```
int[] x = new int[128 * 1024 * 1024];
for (int iter = 0; iter < 5; iter++)
{
    var sw = Stopwatch.StartNew();
    for (int i = 0; i < x.Length; i += 16)
        x[i]++;
    sw.Stop();
    WriteLine(sw.ElapsedMilliseconds);
}
```

Cache-sensitive benchmarks

Let's run a benchmark several times:

```
int[] x = new int[128 * 1024 * 1024];
for (int iter = 0; iter < 5; iter++)
{
    var sw = Stopwatch.StartNew();
    for (int i = 0; i < x.Length; i += 16)
        x[i]++;
    sw.Stop();
    WriteLine(sw.ElapsedMilliseconds);
}
```

```
176 // not warmed
81  // still not warmed
62  // the steady state
62  // the steady state
62  // the steady state
```

Warmup is not only about .NET

Branch prediction

```
const int N = 32767;
int[] sorted, unsorted; // random numbers [0..255]
private static int Sum(int[] data)
{
  int sum = 0;
  for (int i = 0; i < N; i++)
    if (data[i] >= 128)
      sum += data[i];
  return sum;
}
```

```
[Benchmark]
public int Sorted()
{
  return Sum(sorted);
}
```

```
[Benchmark]
public int Unsorted()
{
  return Sum(unsorted);
}
```

Branch prediction

```
const int N = 32767;
int[] sorted, unsorted; // random numbers [0..255]
private static int Sum(int[] data)
{
  int sum = 0;
  for (int i = 0; i < N; i++)
    if (data[i] >= 128)
      sum += data[i];
  return sum;
}
```

```
[Benchmark]
public int Sorted()
{
  return Sum(sorted);
}
```

```
[Benchmark]
public int Unsorted()
{
  return Sum(unsorted);
}
```

	Sorted	Unsorted
LegacyJIT-x86	$\approx 20\mu s$	$\approx 139\mu s$

A bad benchmark

```
var sw1 = Stopwatch.StartNew();
Foo();
sw1.Stop();
var sw2 = Stopwatch.StartNew();
Bar();
sw2.Stop();
```

A bad benchmark

```
var sw1 = Stopwatch.StartNew();
Foo();
sw1.Stop();
var sw2 = Stopwatch.StartNew();
Bar();
sw2.Stop();
```

In general case, you should run each benchmark in his own process. Remember about:

- Interface method dispatch
- Garbage collector and autotuning
- Conditional jitting

Interface method dispatch

```
private interface IInc {
  double Inc(double x);
}
private class Foo : IInc {
  double Inc(double x) => x + 1;
}
private class Bar : IInc {
  double Inc(double x) => x + 1;
}
private double Run(IInc inc) {
  double sum = 0;
  for (int i = 0; i < 1001; i++)
    sum += inc.Inc(0);
  return sum;
}
```

```
// Which method is faster?

[Benchmark]
public double FooFoo() {
  var foo1 = new Foo();
  var foo2 = new Foo();
  return Run(foo1) + Run(foo2);
}

[Benchmark]
public double FooBar() {
  var foo = new Foo();
  var bar = new Bar();
  return Run(foo) + Run(bar);
}
```

Interface method dispatch

```
private interface IInc {
  double Inc(double x);
}
private class Foo : IInc {
  double Inc(double x) => x + 1;
}
private class Bar : IInc {
  double Inc(double x) => x + 1;
}
private double Run(IInc inc) {
  double sum = 0;
  for (int i = 0; i < 1001; i++)
    sum += inc.Inc(0);
  return sum;
}
```

```
// Which method is faster?

[Benchmark]
public double FooFoo() {
  var foo1 = new Foo();
  var foo2 = new Foo();
  return Run(foo1) + Run(foo2);
}

[Benchmark]
public double FooBar() {
  var foo = new Foo();
  var bar = new Bar();
  return Run(foo) + Run(bar);
}
```

	FooFoo	FooBar
LegacyJIT-x64	$\approx 5.4\mu s$	$\approx 7.1\mu s$

Tricky inlining

```
[Benchmark]
int Calc() => WithoutStarg(0x11) + WithStarg(0x12);
int WithoutStarg(int value) => value;
int WithStarg(int value) {
  if (value < 0)
    value = -value;
  return value;
}
```

Tricky inlining

```
[Benchmark]
int Calc() => WithoutStarg(0x11) + WithStarg(0x12);
int WithoutStarg(int value) => value;
int WithStarg(int value) {
  if (value < 0)
    value = -value;
  return value;
}
```

LegacyJIT-x86	LegacyJIT-x64	RyuJIT-x64
≈1.7ns	0	≈1.7ns

Tricky inlining

```
[Benchmark]
int Calc() => WithoutStarg(0x11) + WithStarg(0x12);
int WithoutStarg(int value) => value;
int WithStarg(int value) {
  if (value < 0)
    value = -value;
  return value;
}
```

LegacyJIT-x86	LegacyJIT-x64	RyuJIT-x64
≈1.7ns	0	≈1.7ns

```
; LegacyJIT-x64 : Inlining succeeded
mov         ecx,23h
ret
```

Tricky inlining

```
[Benchmark]
int Calc() => WithoutStarg(0x11) + WithStarg(0x12);
int WithoutStarg(int value) => value;
int WithStarg(int value) {
  if (value < 0)
    value = -value;
  return value;
}
```

LegacyJIT-x86	LegacyJIT-x64	RyuJIT-x64
≈1.7ns	0	≈1.7ns

```
; LegacyJIT-x64 : Inlining succeeded
mov         ecx,23h
ret
```

```
// RyuJIT-x64   : Inlining failed
// Inline expansion aborted due to opcode
// [06] OP_starg.s in method
// Program:WithStarg(int):int:this
```

SIMD

```
struct MyVector // Copy-pasted from System.Numerics.Vector4
{
    public float X, Y, Z, W;
    public MyVector(float x, float y, float z, float w)
    {
        X = x; Y = y; Z = z; W = w;
    }
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static MyVector operator *(MyVector left, MyVector right)
    {
        return new MyVector(left.X * right.X, left.Y * right.Y,
                            left.Z * right.Z, left.W * right.W);
    }
}
Vector4      vector1,   vector2,   vector3;
MyVector   myVector1, myVector2, myVector3;
[Benchmark] void MyMul() => myVector3 = myVector1 * myVector2;
[Benchmark] void BclMul() => vector3 = vector1 * vector2;
```

SIMD

```
struct MyVector // Copy-pasted from System.Numerics.Vector4
{
    public float X, Y, Z, W;
    public MyVector(float x, float y, float z, float w)
    {
        X = x; Y = y; Z = z; W = w;
    }
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public static MyVector operator *(MyVector left, MyVector right)
    {
        return new MyVector(left.X * right.X, left.Y * right.Y,
                            left.Z * right.Z, left.W * right.W);
    }
}
Vector4      vector1,   vector2,   vector3;
MyVector   myVector1, myVector2, myVector3;
[Benchmark] void MyMul() => myVector3 = myVector1 * myVector2;
[Benchmark] void BclMul() => vector3 = vector1 * vector2;
```

	LegacyJIT-x64	RyuJIT-x64
MyMul	≈12.9ns	≈2.5ns
BclMul	≈12.9ns	≈0.2ns

How so?

	LegacyJIT-x64	RyuJIT-x64
MyMul	≈12.9ns	≈2.5ns
BclMul	≈12.9ns	≈0.2ns

```
; LegacyJIT-x64

; MyMul, BclMul: Naïve SSE
; ...
movss       xmm3,dword ptr [rsp+40h]
mulss       xmm3,dword ptr [rsp+30h]
movss       xmm2,dword ptr [rsp+44h]
mulss       xmm2,dword ptr [rsp+34h]
movss       xmm1,dword ptr [rsp+48h]
mulss       xmm1,dword ptr [rsp+38h]
movss       xmm0,dword ptr [rsp+4Ch]
mulss       xmm0,dword ptr [rsp+3Ch]
xor         eax,eax
mov         qword ptr [rsp],rax
mov         qword ptr [rsp+8],rax
lea         rax,[rsp]
movss       dword ptr [rax],xmm3
movss       dword ptr [rax+4],xmm2
; ...
```

```
; RyuJIT-x64

; MyMul: Naïve AVX
; ...
vmulss      xmm0,xmm0,xmm4
vmulss      xmm1,xmm1,xmm5
vmulss      xmm2,xmm2,xmm6
vmulss      xmm3,xmm3,xmm7
; ...

; BclMul: Smart AVX intrinsic
vmovupd     xmm0,xmmword ptr [rcx+8]
vmovupd     xmm1,xmmword ptr [rcx+18h]
vmulps      xmm0,xmm0,xmm1
vmovupd     xmmword ptr [rcx+28h],xmm0
```

Let's calculate some square roots

```
[Benchmark]
double Sqrt13() =>
    Math.Sqrt(1) + Math.Sqrt(2) + Math.Sqrt(3) + /* ... */
    + Math.Sqrt(13);
```

VS

```
[Benchmark]
double Sqrt14() =>
    Math.Sqrt(1) + Math.Sqrt(2) + Math.Sqrt(3) + /* ... */
    + Math.Sqrt(13) + Math.Sqrt(14);
```

Let's calculate some square roots

```
[Benchmark]
double Sqrt13() =>
    Math.Sqrt(1) + Math.Sqrt(2) + Math.Sqrt(3) + /* ... */
    + Math.Sqrt(13);
```

VS

```
[Benchmark]
double Sqrt14() =>
    Math.Sqrt(1) + Math.Sqrt(2) + Math.Sqrt(3) + /* ... */
    + Math.Sqrt(13) + Math.Sqrt(14);
```

	RyuJIT-x64*
Sqrt13	≈91ns
Sqrt14	0 ns

* *Can be changed in future versions, see github.com/dotnet/coreclr/issues/987*

RyuJIT-x64, Sqrt13

```
vsqrtsd     xmm0,xmm0,mmword ptr [7FF94F9E4D28h]
vsqrtsd     xmm1,xmm0,mmword ptr [7FF94F9E4D30h]
vaddsd      xmm0,xmm0,xmm1
vsqrtsd     xmm1,xmm0,mmword ptr [7FF94F9E4D38h]
vaddsd      xmm0,xmm0,xmm1
vsqrtsd     xmm1,xmm0,mmword ptr [7FF94F9E4D40h]
vaddsd      xmm0,xmm0,xmm1
; A lot of vqrtsd and vaddsd instructions
; ...
vsqrtsd     xmm1,xmm0,mmword ptr [7FF94F9E4D88h]
vaddsd      xmm0,xmm0,xmm1
ret
```

RyuJIT-x64, Sqrt14

```
vmovsd      xmm0,qword ptr [7FF94F9C4C80h] ; Const
ret
```

Big expression tree

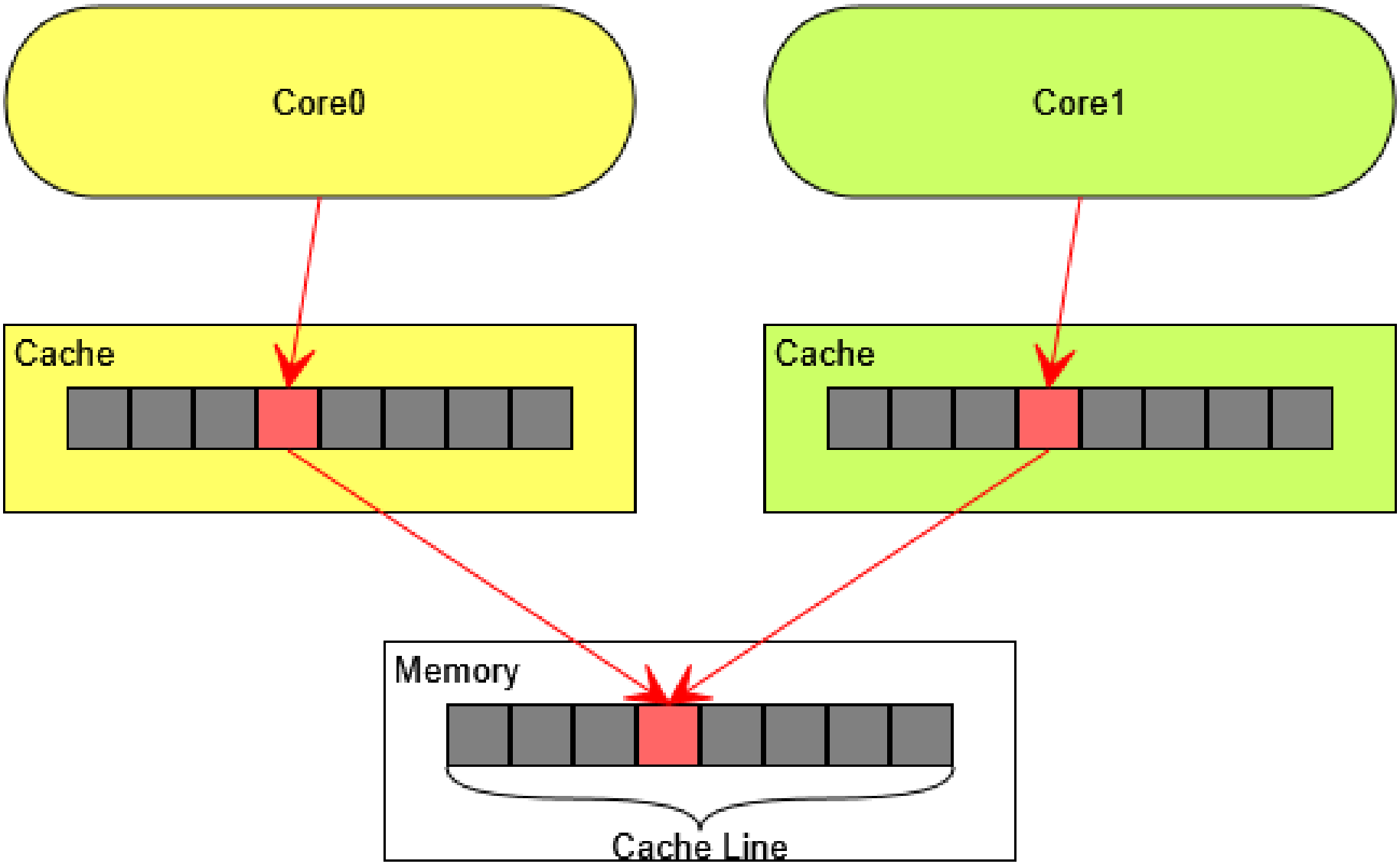
```
*  stmtExpr  void  (top level) (IL 0x000...  ???)
|     /--*  mathFN    double sqrt
|     |  \--*  dconst    double 13.000000000000000
|  /--*  +         double
|  |  |  /--*  mathFN    double sqrt
|  |  |  |  \--*  dconst    double 12.000000000000000
|  |  \--*  +         double
|  |     |  /--*  mathFN    double sqrt
|  |     |  |  \--*  dconst    double 11.000000000000000
|  |     \--*  +         double
|  |        |  /--*  mathFN    double sqrt
|  |        |  |  \--*  dconst    double 10.000000000000000
|  |        \--*  +         double
|  |           |  /--*  mathFN    double sqrt
|  |           |  |  \--*  dconst    double 9.0000000000000000
|  |           \--*  +         double
|  |              |  /--*  mathFN    double sqrt
|  |              |  |  \--*  dconst    double 8.0000000000000000
|  |              \--*  +         double
|  |                 |  /--*  mathFN    double sqrt
|  |                 |  |  \--*  dconst    double 7.0000000000000000
|  |                 \--*  +         double
|  |                    |  /--*  mathFN    double sqrt
|  |                    |  |  \--*  dconst    double 6.0000000000000000
|  |                    \--*  +         double
|  |                       |  /--*  mathFN    double sqrt
|  |                       |  |  \--*  dconst    double 5.0000000000000000
// ...
```

Constant folding in action

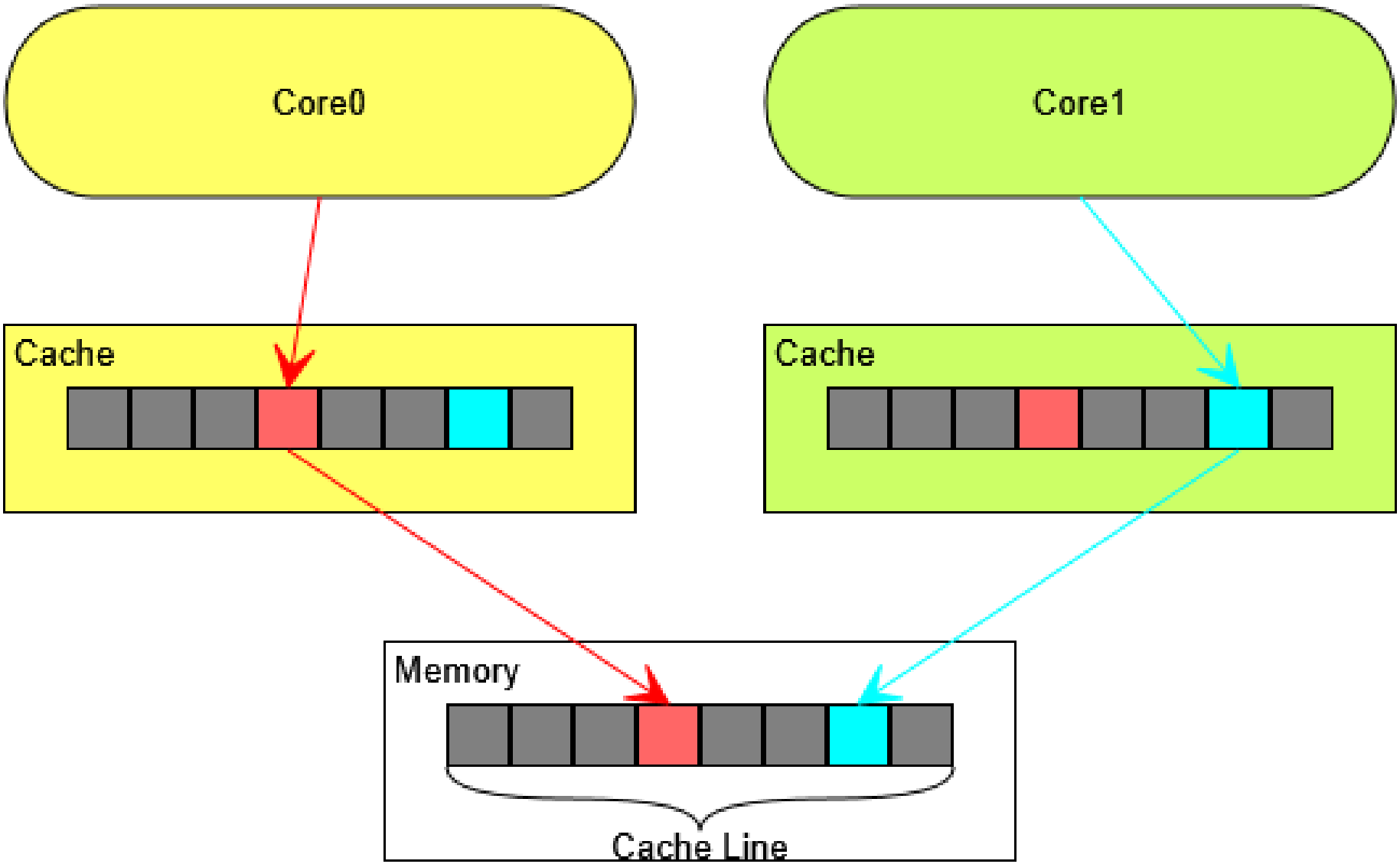
```
N001 [000001]   dconst    1.0000000000000000 => $c0 {DblCns[1.000000]}
N002 [000002]   mathFN    => $c0 {DblCns[1.000000]}
N003 [000003]   dconst    2.0000000000000000 => $c1 {DblCns[2.000000]}
N004 [000004]   mathFN    => $c2 {DblCns[1.414214]}
N005 [000005]   +         => $c3 {DblCns[2.414214]}
N006 [000006]   dconst    3.0000000000000000 => $c4 {DblCns[3.000000]}
N007 [000007]   mathFN    => $c5 {DblCns[1.732051]}
N008 [000008]   +         => $c6 {DblCns[4.146264]}
N009 [000009]   dconst    4.0000000000000000 => $c7 {DblCns[4.000000]}
N010 [000010]   mathFN    => $c1 {DblCns[2.000000]}
N011 [000011]   +         => $c8 {DblCns[6.146264]}
N012 [000012]   dconst    5.0000000000000000 => $c9 {DblCns[5.000000]}
N013 [000013]   mathFN    => $ca {DblCns[2.236068]}
N014 [000014]   +         => $cb {DblCns[8.382332]}
N015 [000015]   dconst    6.0000000000000000 => $cc {DblCns[6.000000]}
N016 [000016]   mathFN    => $cd {DblCns[2.449490]}
N017 [000017]   +         => $ce {DblCns[10.831822]}
N018 [000018]   dconst    7.0000000000000000 => $cf {DblCns[7.000000]}
N019 [000019]   mathFN    => $d0 {DblCns[2.645751]}
N020 [000020]   +         => $d1 {DblCns[13.477573]}
...
```

Concurrency

True sharing



False sharing



False sharing in action

```
// It's an extremely naïve benchmark
// Don't try this at home
int[] x = new int[1024];
void Inc(int p) {
    for (int i = 0; i < 10000001; i++)
        x[p]++;
}
void Run(int step) {
    var sw = Stopwatch.StartNew();
    Task.WaitAll(
        Task.Factory.StartNew(() => Inc(0 * step)),
        Task.Factory.StartNew(() => Inc(1 * step)),
        Task.Factory.StartNew(() => Inc(2 * step)),
        Task.Factory.StartNew(() => Inc(3 * step)));
    WriteLine(sw.ElapsedMilliseconds);
}
```

False sharing in action

```
// It's an extremely naïve benchmark
// Don't try this at home
int[] x = new int[1024];
void Inc(int p) {
    for (int i = 0; i < 10000001; i++)
        x[p]++;
}
void Run(int step) {
    var sw = Stopwatch.StartNew();
    Task.WaitAll(
        Task.Factory.StartNew(() => Inc(0 * step)),
        Task.Factory.StartNew(() => Inc(1 * step)),
        Task.Factory.StartNew(() => Inc(2 * step)),
        Task.Factory.StartNew(() => Inc(3 * step)));
    WriteLine(sw.ElapsedMilliseconds);
}
```

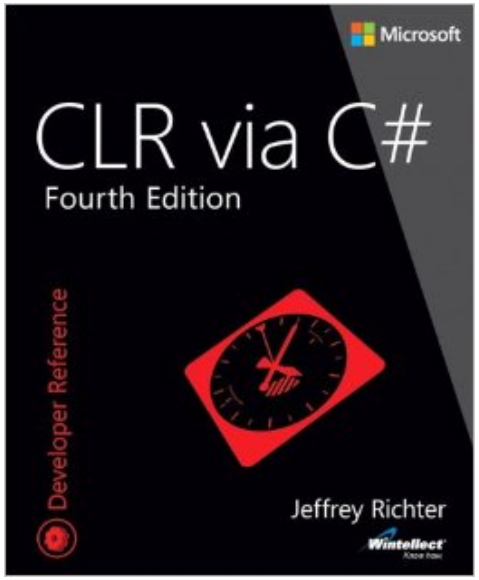
Run(1)	Run(256)
≈400ms	≈150ms

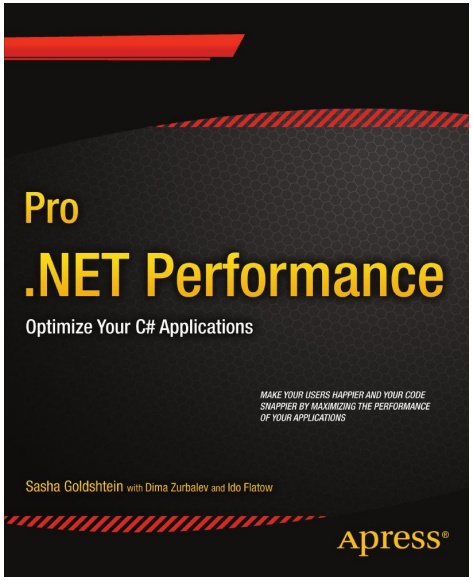
Conclusion: Benchmarking is hard

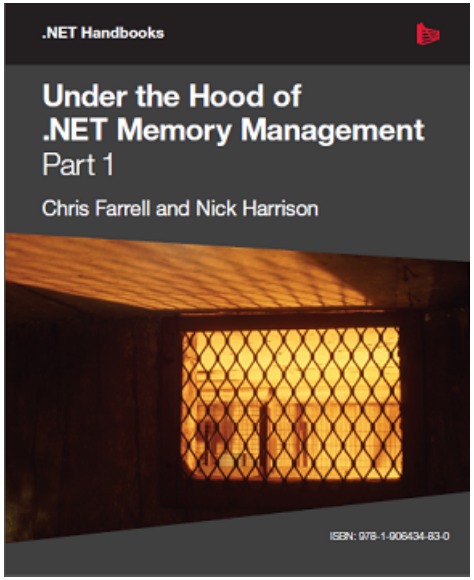
Anon et al., "A Measure of Transaction Processing Power"

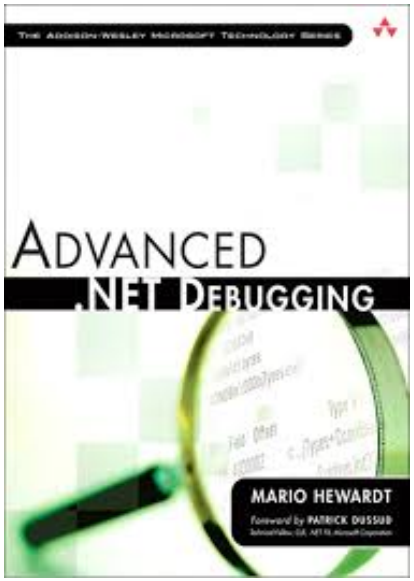
There are lies, damn lies and then there are performance measures.

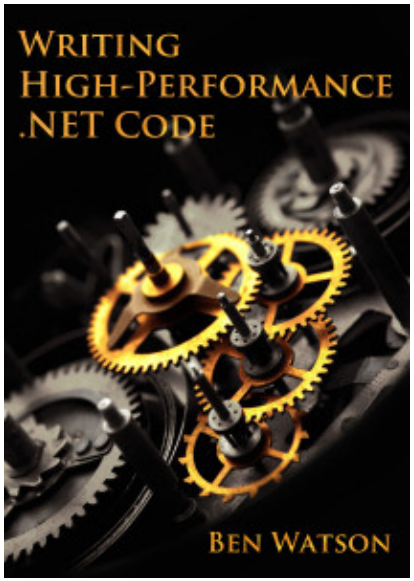
Some good books

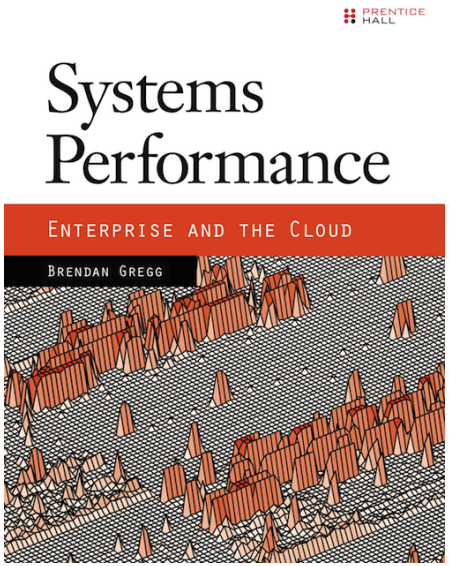












Questions?

Andrey Akinshin
http://aakinshin.net
https://github.com/AndreyAkinshin
https://twitter.com/andrey_akinshin
andrey.akinshin@gmail.com