POWERFUL BENCHMARKING IN .NET

Andrey Akinshin, Adam Sitnik

Why performance is important?

- Responsiveness – customer experience $
- Scalability – scale and earn more $
- Capacity – optimize and save more $
- Power – CPU uses power, which costs $
- Heat – CPU generates heat, contributes to global warming!

Without data you're just another person with an opinion

— W. Edwards Deming, a data scientist

The worst performance optimization is one that is based on incorrect measurements; unfortunately, manual benchmarking often leads into this trap.

- Sasha Goldshtein, Performance Guru

Benchmark? Profiler?

„In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it”

Wikipedia

„In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.”

Wikipedia

PerfDotNet / **BenchmarkDotNet**

Unwatch 26 | Unstar 187 | Fork 18

.NET library for benchmarking https://www.nuget.org/packages/BenchmarkDotNet/ — Edit

161 commits | 1 branch | 9 releases | 9 contributors

BenchmarkDotNet 0.7.8

Lightweight .NET library for benchmarking

To install BenchmarkDotNet, run the following command in the Package Manager Console

```
PM> Install-Package BenchmarkDotNet
```

2 182
Downloads

229
Downloads of v 0.7.8

2015-10-01
Last published

Owners


jon skeet


AndreyAkinshin

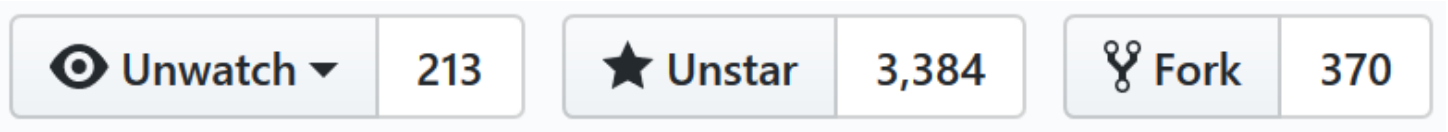

MattWarren

The Contributors

AndreyAkinshin	576 commits	74,782 ++	52,805 --	#1	adamsitnik	549 commits	127,499 ++	113,874 --	#2
mattwarren	108 commits	23,929 ++	6,454 --	#3	alinasmirnova	27 commits	4,627 ++	2,939 --	#4
Ky7m	21 commits	913 ++	518 --	#5	ig-sinicyn	21 commits	6,364 ++	2,604 --	#6
Rizzen	8 commits	1,159 ++	210 --	#7	epeshk	8 commits	112 ++	70 --	#8
redknightlois	8 commits	638 ++	37 --	#9	morgan-kn	7 commits	1,805 ++	346 --	#10
Teknikaali	7 commits	1,861 ++	141 --	#11	lahma	6 commits	209 ++	63 --	#12
lukasz-pyrzyk	6 commits	597 ++	85 --	#13	gigi81	6 commits	272 ++	43 --	#14
dlemstra	5 commits	613 ++	53 --	#15	FransBouma	5 commits	15,786 ++	14,580 --	#16
mfilippov	4 commits	46 ++	257 --	#17	AmadeusW	4 commits	436 ++	63 --	#18
roji	4 commits	1,185 ++	98 --	#19	ppanyukov	4 commits	228 ++	63 --	#20
stevedesmond-ca	3 commits	71 ++	10 --	#21	svick	3 commits	66 ++	55 --	#22
mtschneiders	3 commits	9 ++	9 --	#23	wojtpl2	2 commits	288 ++	7 --	#24
bgrainger	2 commits	100 ++	2 --	#25	facundofarias	2 commits	3 ++	3 --	#26
Tornhoof	2 commits	2 ++	2 --	#27	ltrzesniewski	2 commits	785 ++	255 --	#28
dmitry-ra	2 commits	71 ++	71 --	#29	shoelzer	2 commits	48 ++	70 --	#30
ENikS	2 commits	21 ++	11 --	#31	Chrisgozd	2 commits	186 ++	3 --	#32
GeorgePlotnikov	2 commits	356 ++	2 --	#33	ipjohnson	2 commits	666 ++	40 --	#34
cdmihai	2 commits	20 ++	10 --	#35	alexandrnikitin	2 commits	154 ++	2 --	#36
krk	2 commits	38 ++	1 --	#37	gsomix	1 commit	439 ++	20 --	#38
xavero	1 commit	2 ++	2 --	#39	NRKirby	1 commit	1 ++	1 --	#40
YohDeadfall	1 commit	1 ++	1 --	#41	fredeil	1 commit	2 ++	2 --	#42
Caballero77	1 commit	32 ++	2 --	#43	agocke	1 commit	10 ++	1 --	#44
houseofcat	1 commit	17 ++	12 --	#45	IanKemp	1 commit	7 ++	2 --	#46
afmorris	1 commit	6 ++	1 --	#47	paulness	1 commit	8 ++	0 --	#48
MishaHusiuk	1 commit	2 ++	2 --	#49	dfederm	1 commit	2 ++	2 --	#50
Matthew-Bonner	1 commit	6 ++	6 --	#51	ScottHutchinson	1 commit	33 ++	2 --	#52
nietras	1 commit	14 ++	3 --	#53	onionhammer	1 commit	2 ++	2 --	#54
benjamin-hodgson	1 commit	2 ++	2 --	#55	AlekseiKudelia	1 commit	2 ++	17 --	#56
eerhardt	1 commit	4 ++	1 --	#57	cincuranet	1 commit	2 ++	2 --	#58
rolshevsky	1 commit	27 ++	28 --	#59	jawn	1 commit	4 ++	4 --	#60
pentp	1 commit	1 ++	1 --	#61	aidmsu	1 commit	1 ++	1 --	#62
smitpatel	1 commit	3 ++	2 --	#63	aarondandy	1 commit	6 ++	6 --	#64
davkean	1 commit	2 ++	2 --	#65	RichLinnell	1 commit	7 ++	3 --	#66
mmayr-at	1 commit	2 ++	2 --	#67	factormystic	1 commit	1 ++	1 --	#68
arthrp	1 commit	2 ++	2 --	#69	DenisIstomin	1 commit	202 ++	29 --	#70
russcam	1 commit	71 ++	1 --	#71	JohanLarsson	1 commit	3 ++	2 --	#72
goldshtn	1 commit	73 ++	5,416 --	#73	cloudRoutine	1 commit	5 ++	11 --	#74
ForNeVeR	1 commit	1 ++	1 --	#75	vkkoshelev	1 commit	1 ++	1 --	#76
NN---	1 commit	2 ++	2 --	#77	mijay	1 commit	173 ++	0 --	#78

What is BenchmarkDotNet?

*„**BenchmarkDotNet** is a powerful .NET library for benchmarking."*



Kestrel SignalR Entity Framework F# Orleans

Elasticsearch Dapper ImageSharp RavenDB NodaTime

Sample

```
public class ParsingBenchmarks
{
    [Benchmark]
    public int ParseInt() => int.Parse("123456789");
}

void Main(string[] args)
    => BenchmarkRunner.Run<ParsingBenchmarks>();
```

Sample Results

```
ParsingBenchmarks.ParseInt: DefaultJob
Runtime = .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT; GC = Concurrent Workstation
Mean = 99.9949 ns, StdErr = 0.0912 ns (0.09%); N = 13, StdDev = 0.3290 ns
Min = 99.6271 ns, Q1 = 99.7953 ns, Median = 99.9093 ns, Q3 = 100.1618 ns, Max = 100.8099 ns
IQR = 0.3664 ns, LowerFence = 99.2456 ns, UpperFence = 100.7114 ns
ConfidenceInterval = [99.6009 ns; 100.3889 ns] (CI 99.9%), Margin = 0.3940 ns (0.39% of Mean)
Skewness = 1.15, Kurtosis = 3.36, MValue = 2
-------------------- Histogram --------------------
[99.505 ns ; 100.932 ns) | @@@@@@@@@@@@@
---------------------------------------------------

// * Summary *

BenchmarkDotNet=v0.11.1.817-nightly, OS=Windows 10.0.17134.376 (1803/April2018Update/Redstone4)
Intel Core i7-5557U CPU 3.10GHz (Broadwell), 1 CPU, 4 logical and 2 physical cores
Frequency=3027349 Hz, Resolution=330.3220 ns, Timer=TSC
.NET Core SDK=2.1.403
  [Host]     : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT
  DefaultJob : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT


   Method |     Mean |     Error |    StdDev |
--------- |---------:|----------:|----------:|
 ParseInt | 99.99 ns | 0.3940 ns | 0.3290 ns |

// * Hints *
Outliers
  ParsingBenchmarks.ParseInt: Default -> 2 outliers were removed

// * Legends *
  Mean   : Arithmetic mean of all measurements
  Error  : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  1 ns   : 1 Nanosecond (0.000000001 sec)
```

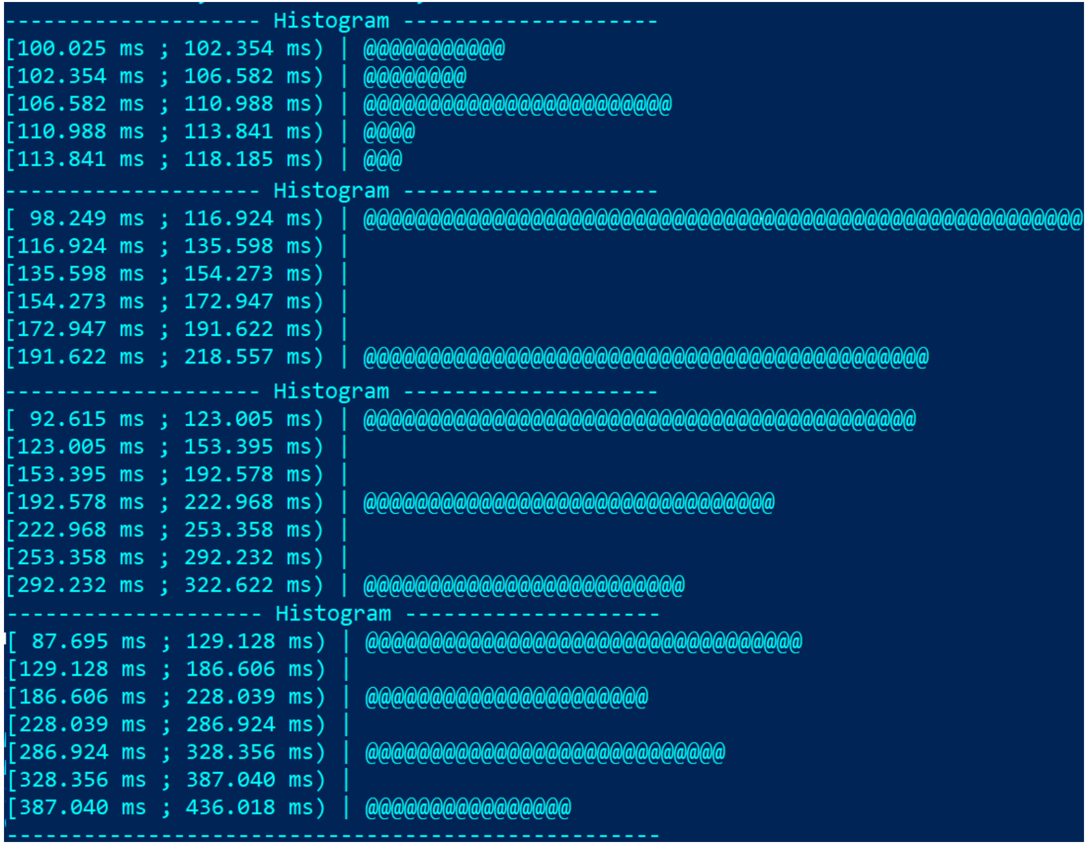
Statistics

- Min, Lower Fence, Q1, Median, Mean, Q3, Upper Fence, Max, Interquartile Range, Outliers
- Standard Error, Variance, Standard Deviation
- Skewness, Kurtosis
- Confidence Intervals
- Percentiles (P0, P25, P50, P67, P80, P85, P90, P95, P100)
- Statistical tests: Welch's t-test, Mann–Whitney U test

Multimodal distribution

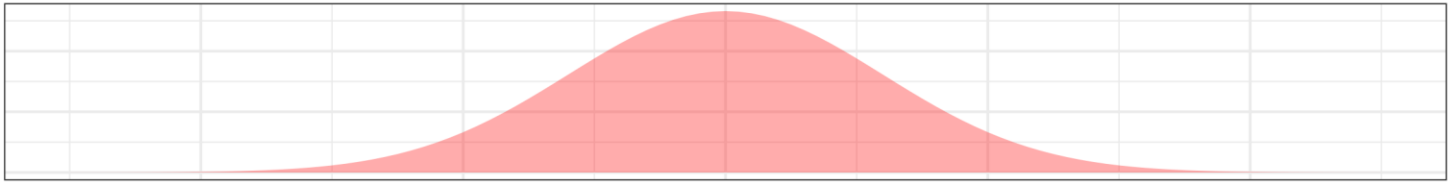
```
[MValueColumn]
[SimpleJob(RunStrategy.Throughput, 1, 0, -1, 1, "MainJob")]
public class IntroMultimodal
{
    private readonly Random rnd = new Random(42);

    private void Multimodal(int n) => Thread.Sleep((rnd.Next(n) + 1) * 100);

    [Benchmark]
    public void Unimodal() => Multimodal(1);

    [Benchmark]
    public void Bimodal() => Multimodal(2);

    [Benchmark]
    public void Trimodal() => Multimodal(3);

    [Benchmark]
    public void Quadrimodal() => Multimodal(4);
}
```

Histogram

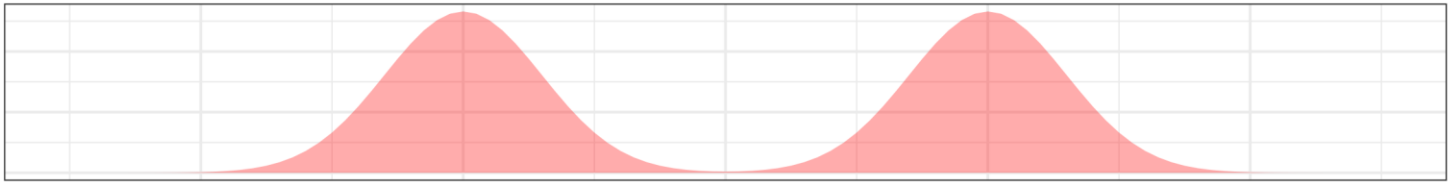


```
-------------------- Histogram --------------------
[100.025 ms ; 102.354 ms) | @@@@@@@@@@@
[102.354 ms ; 106.582 ms) | @@@@@@@@
[106.582 ms ; 110.988 ms) | @@@@@@@@@@@@@@@@@@@@@@@@
[110.988 ms ; 113.841 ms) | @@@@
[113.841 ms ; 118.185 ms) | @@@
-------------------- Histogram --------------------
[ 98.249 ms ; 116.924 ms) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
[116.924 ms ; 135.598 ms) |
[135.598 ms ; 154.273 ms) |
[154.273 ms ; 172.947 ms) |
[172.947 ms ; 191.622 ms) |
[191.622 ms ; 218.557 ms) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-------------------- Histogram --------------------
[ 92.615 ms ; 123.005 ms) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
[123.005 ms ; 153.395 ms) |
[153.395 ms ; 192.578 ms) |
[192.578 ms ; 222.968 ms) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
[222.968 ms ; 253.358 ms) |
[253.358 ms ; 292.232 ms) |
[292.232 ms ; 322.622 ms) | @@@@@@@@@@@@@@@@@@@@@@@@@
-------------------- Histogram --------------------
[ 87.695 ms ; 129.128 ms) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
[129.128 ms ; 186.606 ms) |
[186.606 ms ; 228.039 ms) | @@@@@@@@@@@@@@@@@@@@@@
[228.039 ms ; 286.924 ms) |
[286.924 ms ; 328.356 ms) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@
[328.356 ms ; 387.040 ms) |
[387.040 ms ; 436.018 ms) | @@@@@@@@@@@@@@@@
---------------------------------------------------
```

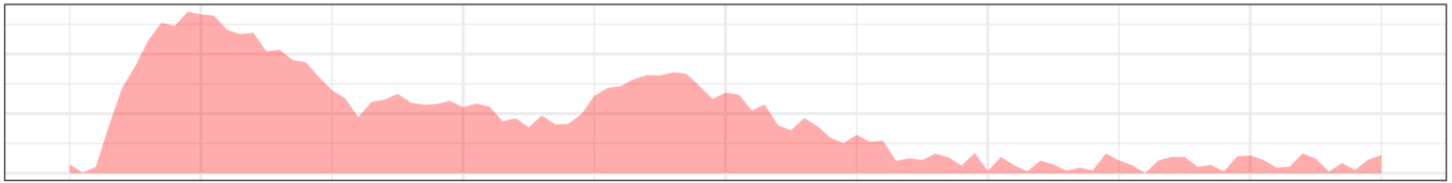
Expectation and reality



(A) Unimodal distribution

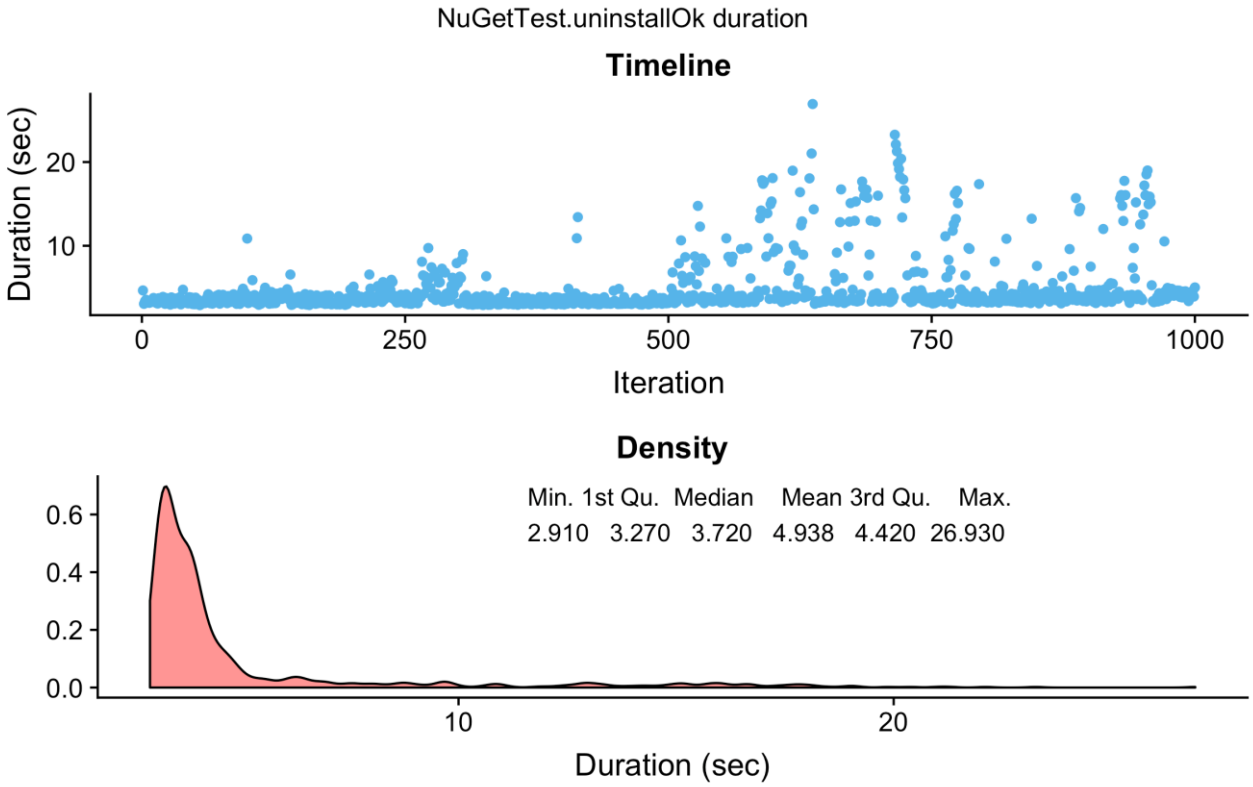


(B) Bimodal distribution

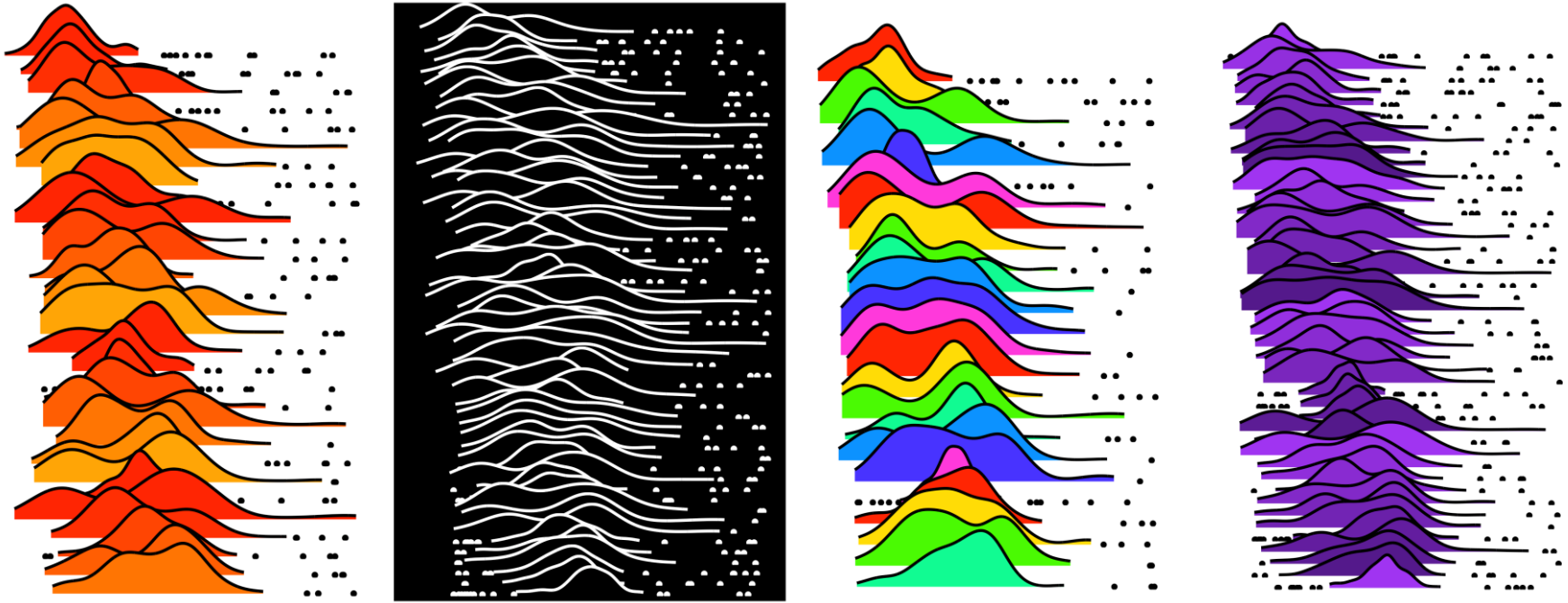


(C) Performance distribution

Outliers

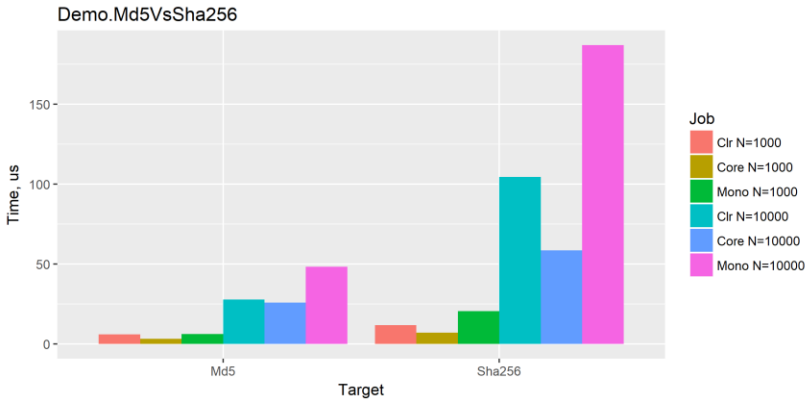

NuGetTest.uninstallOk duration
Timeline
Duration (sec)
20
10
0
250
500
750
1000
Iteration
Density
Min. 1st Qu. Median Mean 3rd Qu. Max.
2.910 3.270 3.720 4.938 4.420 26.930
0.6
0.4
0.2
0.0
10
20
Duration (sec)

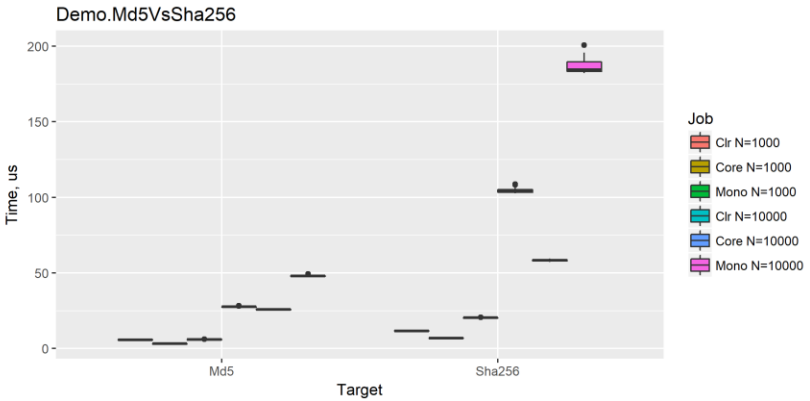
Frequency trails

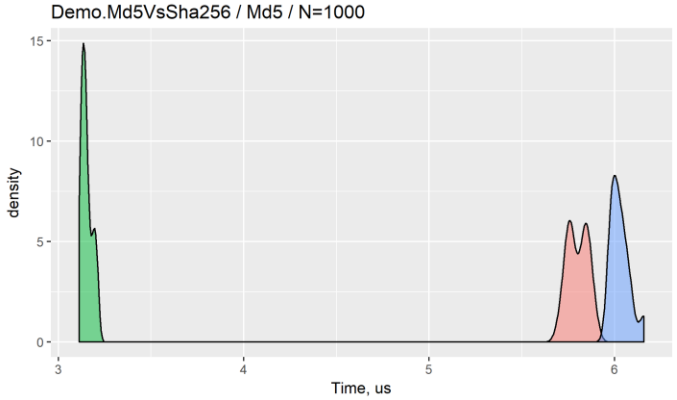


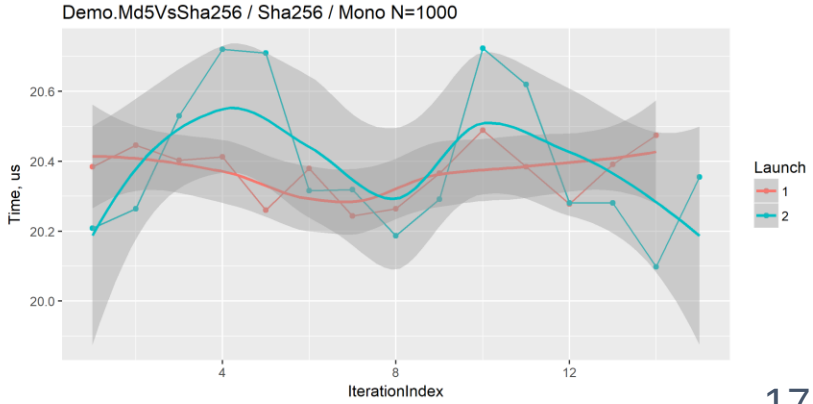
See also: http://www.brendangregg.com/FrequencyTrails/intro.html

RPlot Sample









Exporters

- HTML
- Markdown: GitHub, StackOverflow
- CSV
- RPlot (requires R)
- XML
- JSON

```
[AsciiDocExporter]
[CsvExporter]
[CsvMeasurementsExporter]
[HtmlExporter]
[PlainExporter]
[RPlotExporter]
[JsonExporterAttribute.Brief]
[JsonExporterAttribute.BriefCompressed]
[JsonExporterAttribute.Full]
[JsonExporterAttribute.FullCompressed]
[MarkdownExporterAttribute.Default]
[MarkdownExporterAttribute.GitHub]
[MarkdownExporterAttribute.StackOverflow]
[MarkdownExporterAttribute.Atlassian]
[XmlExporterAttribute.Brief]
[XmlExporterAttribute.BriefCompressed]
[XmlExporterAttribute.Full]
[XmlExporterAttribute.FullCompressed]
public class IntroExporters
```

.\BenchmarkDotNet.Artifacts\results

Sitnik > source > repos > BdnDemo > BdnDemo > BenchmarkDotNet.Artifacts > results

Search results

Name	Date modified	Type	Size
BdnDemo.AccurateAllocations-report.csv	11/4/2018 2:29 AM	Microsoft Excel Com...	1 KB
BdnDemo.AccurateAllocations-report.html	11/4/2018 2:29 AM	HTML File	2 KB
BdnDemo.AccurateAllocations-report-github.md	11/4/2018 2:29 AM	MD File	1 KB
BdnDemo.IntroMultimodal-report.csv	10/26/201		
BdnDemo.IntroMultimodal-report.html	10/26/201		
BdnDemo.IntroMultimodal-report-github.md	10/26/201		
BdnDemo.ListBenchmarks-report.csv	10/26/201		
BdnDemo.ListBenchmarks-report.html	10/26/201		
BdnDemo.ListBenchmarks-report-github.md	10/26/201		
BdnDemo.Md5VsSha256_Md5_DefaultJob-asm....	10/24/201		
BdnDemo.Md5VsSha256_Md5_DefaultJob-asm....	10/24/201		
BdnDemo.Md5VsSha256_Md5_DefaultJob-asm....	10/24/201		
BdnDemo.Md5VsSha256-disassembly-report.ht...	10/24/201		
BdnDemo.Md5VsSha256-report.csv	10/27/201		
BdnDemo.Md5VsSha256-report.html	10/27/201		
BdnDemo.Md5VsSha256-report-github.md	10/27/201		
BdnDemo.OperationsPerInvokeSample_Increm...	10/26/201		
BdnDemo.OperationsPerInvokeSample_Increm...	10/26/201		
BdnDemo.OperationsPerInvokeSample_Increm...	10/26/201		
BdnDemo.OperationsPerInvokeSample_Increm...	10/26/201		
BdnDemo.OperationsPerInvokeSample_Increm...	10/26/201		

C:\Users\adsitnik\source\repos\BdnDemo\BdnDemo\BenchmarkDotNet.Artifacts\results\BdnDemo.Md...

File Edit Search View Encoding Language Settings Tools Macro Run Window ?

IntroDisasm.cs | BdnDemo.AccurateAllocations-report-github.md | BdnDemo.Md5VsSha256-report-github.md

```ini

BenchmarkDotNet=v0.11.1.817-nightly, OS=Windows 10.0.17134.376 (1803/April2018Update/
Intel Core i7-5557U CPU 3.10GHz (Broadwell), 1 CPU, 4 logical and 2 physical cores
Frequency=3027346 Hz, Resolution=330.3223 ns, Timer=TSC
.NET Core SDK=2.1.403
  [Host]     : .NET Core 2.0.7 (CoreCLR 4.6.26328.01, CoreFX 4.6.26403.03), 64bit Ryu
  Job-VPAMDV : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.3221.0
  Job-HUVHDP : .NET Core 2.0.7 (CoreCLR 4.6.26328.01, CoreFX 4.6.26403.03), 64bit Ryu
  Job-QUSOBD : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit Ryu


```

Method	Runtime	Toolchain	Mean	Error	StdDev
Md5	Clr	net46	21.56 us	0.0289 us	0.0241 us
Md5	Core	netcoreapp2.0	20.40 us	0.1329 us	0.1110 us
Md5	Core	netcoreapp2.1	20.35 us	0.0352 us	0.0329 us

BenchmarkSwitcher

```
void Main(string[] args)
    => BenchmarkSwitcher
            .FromAssembly(typeof(Program).Assembly)
            .Run(args);
```

```
PS C:\Projects\performance\src\benchmarks> dotnet run -c Release -f netcoreapp2.1
Available Benchmarks:
  #0 EnumBenchmarks
  #1 EqualityComparerBenchmarks
  #2 ParsingBenchmarks
  #3 StringBenchmarks
  #4 JitBenchmarks


You should select the target benchmark. Please, print a number of a benchmark (e.g. '0') or a benchmark caption (e.g. 'EnumBenchmarks'):
```

Use `--filter` and `--list`!

--list

- --list flat | tree

```
PS C:\Users\adsitnik\source\repos\BdnDemo\BdnDemo> dotnet run -c Release -f netcoreapp2.0 -- --list tree
BdnDemo
 ├─IntroMultimodal
 │  ├─Unimodal
 │  ├─Bimodal
 │  ├─Trimodal
 │  └─Quadrimodal
 ├─ListBenchmarks
 │  ├─Add
 │  └─AddLoop
 ├─Md5VsSha256
 │  ├─Sha256
 │  └─Md5
 └─ParsingBenchmarks
    └─ParseInt
```

How does it work?

- Auto mode (default):
  - Jitting
  - Pilot
  - Overhead Warmup
  - Overhead Actual
  - Workload Warmup
  - Workload Actual
- Specific (configured):
  - Overhead Warmup
  - Overhead Actual
  - Workload Warmup
  - Workload Actual

Jitting

```
OverheadJitting  1: 1 op, 313475.59 ns, 313.4756 us/op
WorkloadJitting  1: 1 op, 2107784.73 ns, 2.1078 ms/op

OverheadJitting  2: 16 op, 741242.59 ns, 46.3277 us/op
WorkloadJitting  2: 16 op, 610104.75 ns, 38.1315 us/op
```

Pilot stage – perfect invocation count

```
WorkloadPilot    1: 16 op, 5615.47 ns, 350.9671 ns/op
WorkloadPilot    2: 32 op, 6606.44 ns, 206.4513 ns/op
WorkloadPilot    3: 64 op, 23452.86 ns, 366.4510 ns/op
WorkloadPilot    4: 128 op, 42941.86 ns, 335.4833 ns/op
WorkloadPilot    5: 256 op, 93150.81 ns, 363.8703 ns/op
WorkloadPilot    6: 512 op, 64743.11 ns, 126.4514 ns/op
WorkloadPilot    7: 1024 op, 148975.23 ns, 145.4836 ns/op
WorkloadPilot    8: 2048 op, 286058.86 ns, 139.6772 ns/op
WorkloadPilot    9: 4096 op, 540737.13 ns, 132.0159 ns/op
WorkloadPilot   10: 8192 op, 953309.31 ns, 116.3708 ns/op
WorkloadPilot   11: 16384 op, 1912564.43 ns, 116.7337 ns/op
WorkloadPilot   12: 32768 op, 3450213.37 ns, 105.2922 ns/op
WorkloadPilot   13: 65536 op, 7242640.34 ns, 110.5139 ns/op
WorkloadPilot   14: 131072 op, 13963041.59 ns, 106.5296 ns/op
WorkloadPilot   15: 262144 op, 28827531.94 ns, 109.9683 ns/op
WorkloadPilot   16: 524288 op, 57801396.54 ns, 110.2474 ns/op
WorkloadPilot   17: 1048576 op, 108772394.59 ns, 103.7334 ns/op
WorkloadPilot   18: 2097152 op, 216061643.37 ns, 103.0262 ns/op
WorkloadPilot   19: 4194304 op, 429615812.38 ns, 102.4284 ns/op
WorkloadPilot   20: 8388608 op, 869214286.16 ns, 103.6184 ns/op
```

Heuristic

```
job.WithInvocationCount(count) or --invocationCount
```

Result = (Result + Overhead) - Overhead

```
OverheadActual   1: 8388608 op, 16477122.39 ns, 1.9642 ns/op
OverheadActual   2: 8388608 op, 16628740.19 ns, 1.9823 ns/op
OverheadActual   3: 8388608 op, 16199982.23 ns, 1.9312 ns/op
OverheadActual   4: 8388608 op, 16220131.87 ns, 1.9336 ns/op
OverheadActual   5: 8388608 op, 16184787.42 ns, 1.9294 ns/op
OverheadActual   6: 8388608 op, 16199982.23 ns, 1.9312 ns/op
OverheadActual   7: 8388608 op, 16763841.90 ns, 1.9984 ns/op
OverheadActual   8: 8388608 op, 16979542.17 ns, 2.0241 ns/op
OverheadActual   9: 8388608 op, 17134463.19 ns, 2.0426 ns/op
OverheadActual  10: 8388608 op, 16771769.62 ns, 1.9994 ns/op
OverheadActual  11: 8388608 op, 16812399.23 ns, 2.0042 ns/op
OverheadActual  12: 8388608 op, 16797865.06 ns, 2.0025 ns/op
OverheadActual  13: 8388608 op, 17373286.00 ns, 2.0711 ns/op
OverheadActual  14: 8388608 op, 16612224.09 ns, 1.9803 ns/op
OverheadActual  15: 8388608 op, 16755914.17 ns, 1.9975 ns/op
```

The Overhead

```
[Benchmark(Description = "Interlocked.Increment(ref int)")]
[Arguments(10)]
public int Increment(ref int arg) => Interlocked.Increment(ref arg);

[Benchmark]
[Arguments(10)]
public int Overhead(ref int arg) => 0;

DefaultConfig.Instance
    .With(Job.Default.WithId("NO Overhead"))
    .With(Job.Default.WithEvaluateOverhead(false).WithId("With Overhead"))
```

The difference

```
BenchmarkDotNet=v0.11.1.817-nightly, OS=Windows 10.0.17134.376 (1803/April2018Update/Redstone4)
Intel Core i7-5557U CPU 3.10GHz (Broadwell), 1 CPU, 4 logical and 2 physical cores
Frequency=3027343 Hz, Resolution=330.3227 ns, Timer=TSC
.NET Core SDK=2.1.403
  [Host]        : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT
  NO Overhead   : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT
  With Overhead : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT
```

Method	Job	EvaluateOverhead	arg	Mean	Error	StdDev
'Interlocked.Increment(ref int)'	NO Overhead	Default	10	4.340 ns	0.0057 ns	0.0048 ns
'Interlocked.Increment(ref int)'	With Overhead	False	10	6.200 ns	0.0073 ns	0.0061 ns

Warmup stage

```
WorkloadWarmup   1: 8388608 op, 854038302.16 ns, 101.8093 ns/op
WorkloadWarmup   2: 8388608 op, 855850118.37 ns, 102.0253 ns/op
WorkloadWarmup   3: 8388608 op, 852839893.91 ns, 101.6664 ns/op
WorkloadWarmup   4: 8388608 op, 871725394.07 ns, 103.9178 ns/op
WorkloadWarmup   5: 8388608 op, 852693230.94 ns, 101.6490 ns/op
WorkloadWarmup   6: 8388608 op, 857685387.45 ns, 102.2441 ns/op
```

```
job.WithWarmupCount(count)
```

or

```
--warmupCount
--minWarmupCount
--maxWarmupCount
```

Heuristic

Actual Workload

```
WorkloadActual   1: 8388608 op, 881260138.82 ns, 105.0544 ns/op
WorkloadActual   2: 8388608 op, 853852330.87 ns, 101.7871 ns/op
WorkloadActual   3: 8388608 op, 852393298.56 ns, 101.6132 ns/op
WorkloadActual   4: 8388608 op, 853952748.76 ns, 101.7991 ns/op
WorkloadActual   5: 8388608 op, 853756867.81 ns, 101.7757 ns/op
WorkloadActual   6: 8388608 op, 855847475.79 ns, 102.0250 ns/op
WorkloadActual   7: 8388608 op, 858974634.24 ns, 102.3978 ns/op
WorkloadActual   8: 8388608 op, 852780105.63 ns, 101.6593 ns/op
WorkloadActual   9: 8388608 op, 854761046.71 ns, 101.8955 ns/op
WorkloadActual  10: 8388608 op, 854717113.88 ns, 101.8902 ns/op
WorkloadActual  11: 8388608 op, 854827111.11 ns, 101.9033 ns/op
WorkloadActual  12: 8388608 op, 855137613.80 ns, 101.9403 ns/op
WorkloadActual  13: 8388608 op, 875801237.32 ns, 104.4036 ns/op
WorkloadActual  14: 8388608 op, 857909676.09 ns, 102.2708 ns/op
WorkloadActual  15: 8388608 op, 862315841.35 ns, 102.7961 ns/op
```

```
job.WithIterationCount(count)
```

Heuristic

Results

```
WorkloadResult   1: 8388608 op, 837191527.42 ns, 99.8010 ns/op
WorkloadResult   2: 8388608 op, 835732495.11 ns, 99.6271 ns/op
WorkloadResult   3: 8388608 op, 837291945.31 ns, 99.8130 ns/op
WorkloadResult   4: 8388608 op, 837096064.36 ns, 99.7896 ns/op
WorkloadResult   5: 8388608 op, 839186672.34 ns, 100.0388 ns/op
WorkloadResult   6: 8388608 op, 842313830.79 ns, 100.4116 ns/op
WorkloadResult   7: 8388608 op, 836119302.18 ns, 99.6732 ns/op
WorkloadResult   8: 8388608 op, 838100243.26 ns, 99.9093 ns/op
WorkloadResult   9: 8388608 op, 838056310.43 ns, 99.9041 ns/op
WorkloadResult  10: 8388608 op, 838166307.66 ns, 99.9172 ns/op
WorkloadResult  11: 8388608 op, 838476810.35 ns, 99.9542 ns/op
WorkloadResult  12: 8388608 op, 841248872.64 ns, 100.2847 ns/op
WorkloadResult  13: 8388608 op, 845655037.90 ns, 100.8099 ns/op
```

```
WorkloadActual   1: 8388608 op, 881260138.82 ns, 105.0544 ns/op
WorkloadActual   2: 8388608 op, 853852330.87 ns, 101.7871 ns/op
WorkloadActual   3: 8388608 op, 852393298.56 ns, 101.6132 ns/op
WorkloadActual   4: 8388608 op, 853952748.76 ns, 101.7991 ns/op
WorkloadActual   5: 8388608 op, 853756867.81 ns, 101.7757 ns/op
WorkloadActual   6: 8388608 op, 855847475.79 ns, 102.0250 ns/op
WorkloadActual   7: 8388608 op, 858974634.24 ns, 102.3978 ns/op
WorkloadActual   8: 8388608 op, 852780105.63 ns, 101.6593 ns/op
WorkloadActual   9: 8388608 op, 854761046.71 ns, 101.8955 ns/op
WorkloadActual  10: 8388608 op, 854717113.88 ns, 101.8902 ns/op
WorkloadActual  11: 8388608 op, 854827111.11 ns, 101.9033 ns/op
WorkloadActual  12: 8388608 op, 855137613.80 ns, 101.9403 ns/op
WorkloadActual  13: 8388608 op, 875801237.32 ns, 104.4036 ns/op
WorkloadActual  14: 8388608 op, 857909676.09 ns, 102.2708 ns/op
WorkloadActual  15: 8388608 op, 862315841.35 ns, 102.7961 ns/op

// AfterActualRun
WorkloadResult   1: 8388608 op, 837191527.42 ns, 99.8010 ns/op
WorkloadResult   2: 8388608 op, 835732495.11 ns, 99.6271 ns/op
WorkloadResult   3: 8388608 op, 837291945.31 ns, 99.8130 ns/op
WorkloadResult   4: 8388608 op, 837096064.36 ns, 99.7896 ns/op
WorkloadResult   5: 8388608 op, 839186672.34 ns, 100.0388 ns/op
WorkloadResult   6: 8388608 op, 842313830.79 ns, 100.4116 ns/op
WorkloadResult   7: 8388608 op, 836119302.18 ns, 99.6732 ns/op
WorkloadResult   8: 8388608 op, 838100243.26 ns, 99.9093 ns/op
WorkloadResult   9: 8388608 op, 838056310.43 ns, 99.9041 ns/op
WorkloadResult  10: 8388608 op, 838166307.66 ns, 99.9172 ns/op
WorkloadResult  11: 8388608 op, 838476810.35 ns, 99.9542 ns/op
WorkloadResult  12: 8388608 op, 841248872.64 ns, 100.2847 ns/op
WorkloadResult  13: 8388608 op, 845655037.90 ns, 100.8099 ns/op
```

```
job.WithOutlierMode(mode)
    or --outliers
```

IsOutlier

The trap

```
public class ListBenchmarks
{
    private List<int> list = new List<int>();

    [Benchmark]
    public void Add() => list.Add(1234);

    [Benchmark]
    public void AddLoop()
    {
        list.Clear();

        for (int i = 0; i < 1000; i++)
            list.Add(1234);
    }
}
```

OOM

```
WorkloadActual  19: 67108864 op, 1442179864.15 ns, 21.4901 ns/op
WorkloadActual  20: 67108864 op, 969300202.92 ns, 14.4437 ns/op
WorkloadActual  21: 67108864 op, 474111508.12 ns, 7.0648 ns/op
WorkloadActual  22: 67108864 op, 390771926.20 ns, 5.8230 ns/op

OutOfMemoryException!
BenchmarkDotNet continues to run additional iterations until desired accuracy level is achieved. It's possible only if the benchmark method doesn't have any side-effects.
If your benchmark allocates memory and keeps it alive, you are creating a memory leak.
You should redesign your benchmark and remove the side-effects. You can use `OperationsPerInvoke`, `IterationSetup` and `IterationCleanup` to do that.
```

Setup & Cleanup

```
public class SetupAndCleanupExample
{
    [GlobalSetup]
    public void GlobalSetup() { }

    [IterationSetup] // sets 1 iteration = 1 invocation
    public void IterationSetup() { }

    [Benchmark]
    public void Benchmark() { }

    [IterationCleanup]
    public void IterationCleanup() { }

    [GlobalCleanup]
    public void GlobalCleanup() { }
}
```

More info

Strategies

- Throughput – default, perfect for microbenchmarks with a steady state
- Monitoring
  - no Pilot stage
  - no Overhead evaluation
  - Outliers remain untouched
  - 1 iteration = 1 benchmark invocation
- ColdStart – no warmup, no pilot stage

Stages: Summary

- Using statistics to get stable results
- Users don’t need to worry about specifying invocation count
- Results don’t contain overhead
- **It takes time to do all of that**
- User can specify invocation/iteration/warmup/target count
- User can customize the heuristic
- Benchmarks should not have side-effects

Inlining

```
[Benchmark(Baseline = true)]
public void OneWay() { /* one way to solve the problem */ }
[Benchmark]
public void AnotherWay() { /* another way to solve the problem */ }
```

What if one of the methods get inlined?

How to prevent inlining without modifying the code?

```
public delegate Span<byte> TargetDelegate();

private TargetDelegate targetDelegate = BenchmarkedMethod;
```

How to minimize loop overhead?

```
private void MainMultiAction(long invokeCount)
{
    for (long i = 0; i < invokeCount; i++)
        targetDelegate();
}
```

```
private void MainMultiAction(long invokeCount)
{
    for (long i = 0; i < invokeCount / unrollFactor; i++)
    {
        targetDelegate(); targetDelegate(); targetDelegate(); targetDelegate();
        targetDelegate(); targetDelegate(); targetDelegate(); targetDelegate();
        targetDelegate(); targetDelegate(); targetDelegate(); targetDelegate();
        targetDelegate(); targetDelegate(); targetDelegate(); targetDelegate();
    }
}
```

```
job.WithUnrollFactor(count) or --unrollFactor
```

More info

How to prevent from Out-of-order execution?

```
private void MainMultiAction(long invokeCount)
{
    for (long i = 0; i < invokeCount / unrollFactor; i++)
    {
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
        consumer.Consume(targetDelegate()); consumer.Consume(targetDelegate());
    }
}
```

Consumer

```
public class Consumer
{
    private volatile byte byteHolder;
    // (more types skipped for brevity)
    private string stringHolder;
    private object objectHolder;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Consume(ulong ulongValue)
       => Volatile.Write(ref ulongHolder, ulongValue);
}
```

Architecture

- Host Process (console app)
  - Generates
  - Builds (Roslyn/dotnet cli)
  - Executes Child Process
- Child Process (console app)
  - **Executes benchmark**
  - Signals events to Host
  - Reports results to Host

See: IToolchain, IGenerator, IBuilder and IExecutor

Why Process-level Isolation?

- We want to have stable and repeatable results
- Order of executing benchmarks should not affect the results
  - Benchmarks can have side effects
  - GC is self-tuning (generation size can change over time)
  - We need a clean CPU cache
  - CLR can apply some optimizations
- [InProcessToolchain] does **not** spawn new process

Generating new project

- Benchmark.not**cs** (customized for every benchmark)
- Benchmark.**csproj**
  - Architecture (Job.Env.Platform)
  - Optimizations: ALWAYS on
- Benchmark**.config -** derives from Host.config file, except of:
  - GC Mode (Job.Env.Gc)
  - JIT: Legacy/RyuJIT/LLVm (Job.Env.Jit, *LLVM only for Mono)
  - & more: GCCpuGroup, gcAllowVeryLargeObjects
- Use [KeepBenchmarkFiles] to see what is generated

See: AppConfigGenerator, Generator, EnvMode, GcMode

Run benchmark for all JITs

```
[Config(typeof(JitsConfig))]
public class MathBenchmarks
{
    private class JitsConfig : ManualConfig
    {
        public JitsConfig()
        {
            Add(Job.Default.With(Jit.LegacyJit).With(Platform.X86).WithId("Legacy x86"));
            Add(Job.Default.With(Jit.LegacyJit).With(Platform.X64).WithId("Legacy x64"));
            Add(Job.Default.With(Jit.RyuJit).With(Platform.X64).WithId("Ryu x64"));
        }
    }

    [Benchmark]
    public double Sqrt14()
        => Math.Sqrt(1) + Math.Sqrt(2) + Math.Sqrt(3) + Math.Sqrt(4) +
           Math.Sqrt(5) + Math.Sqrt(6) + Math.Sqrt(7) + Math.Sqrt(8) +
           Math.Sqrt(9) + Math.Sqrt(10) + Math.Sqrt(11) + Math.Sqrt(12) +
           Math.Sqrt(13) + Math.Sqrt(14);
}
```

LegacyJit vs RyuJit

```
BenchmarkDotNet=v0.10.14.20180425-develop, OS=Windows 10.0.16299.371 (1709/FallCreatorsUpdate/Redstone3)
Intel Core i7-6700 CPU 3.40GHz (Skylake), 1 CPU, 8 logical and 4 physical cores
Frequency=3328125 Hz, Resolution=300.4695 ns, Timer=TSC
  [Host]     : .NET Framework 4.7.1 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2633.0
  Legacy x64 : .NET Framework 4.7.1 (CLR 4.0.30319.42000), 64bit LegacyJIT/clrjit-v4.7.2633.0;compatjit-v4.7.2633.0
  Legacy x86 : .NET Framework 4.7.1 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.7.2633.0
  Ryu x64    : .NET Framework 4.7.1 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2633.0


 Method |        Job |       Jit | Platform |       Mean |      Error |     StdDev |
------- |----------- |---------- |--------- |----------:|----------:|----------:|
 Sqrt14 | Legacy x64 | LegacyJit |      X64 | 65.6634 ns |  0.7894 ns |  0.6998 ns |
 Sqrt14 | Legacy x86 | LegacyJit |      X86 | 12.4520 ns | 13.1436 ns | 18.4255 ns |
 Sqrt14 |    Ryu x64 |    RyuJit |      X64 |  0.0000 ns |  0.0000 ns |  0.0000 ns |

// * Hints *
Outliers
  MathBenchmarks.Sqrt14: Legacy x64 -> 1 outlier  was  removed
  MathBenchmarks.Sqrt14: Legacy x86 -> 1 outlier  was  removed
```

Why 0ns for RyuJIT?!? Is it a bug?

Different GC modes

```
[Config(typeof(GcConfig))]
public class GcBenchmarks
{
    private class GcConfig : ManualConfig
    {
        public GcConfig()
        {
            Add(Job.Default.With(new GcMode { Server = true, Concurrent = true }).WithId("Background Server"));
            Add(Job.Default.With(new GcMode { Server = true, Concurrent = false }).WithId("Server"));
            Add(Job.Default.With(new GcMode { Server = false, Concurrent = true }).WithId("Background Workstation"));
            Add(Job.Default.With(new GcMode { Server = false, Concurrent = false }).WithId("Workstation"));

            Add(MemoryDiagnoser.Default);
        }
    }

    [Benchmark(Description = "new byte[10kB]")]
    public byte[] Allocate() => new byte[10000];
}
```

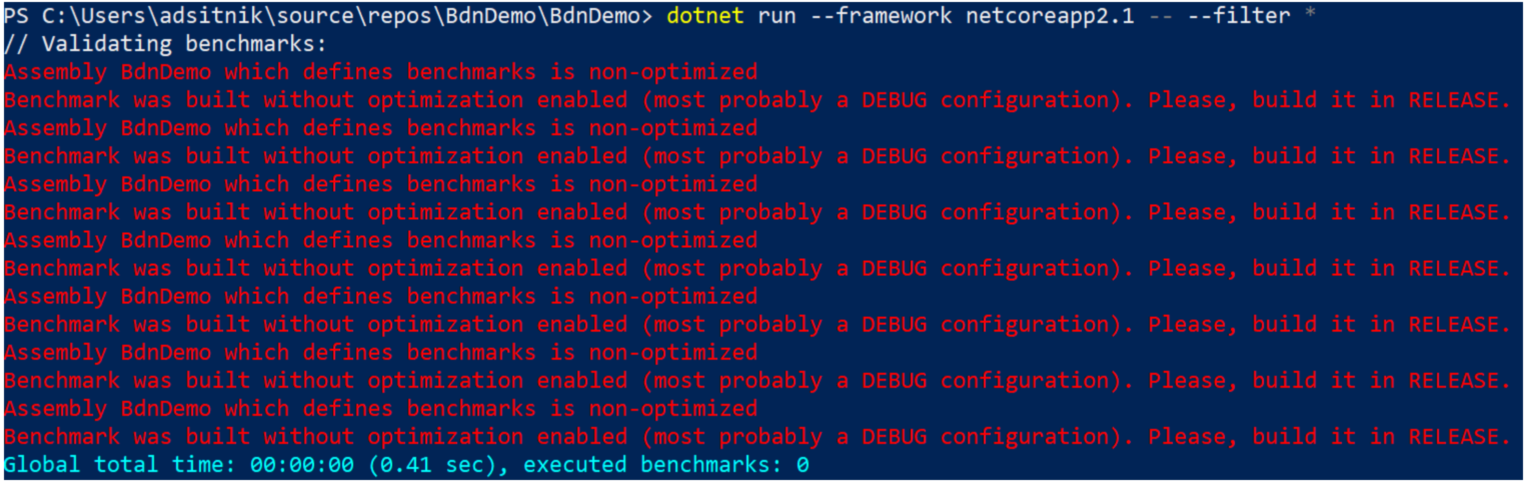
Different GC modes

Method	Job	Concurrent	Server	Mean	Error	StdDev	Median	Gen 0	Gen 1	Allocated
'new byte[10kB]'	Background Server	True	True	779.3 ns	25.996 ns	76.649 ns	722.7 ns	0.1259	-	9.81 KB
'new byte[10kB]'	Background Workstation	True	False	394.4 ns	7.738 ns	7.238 ns	394.6 ns	2.3923	-	9.81 KB
'new byte[10kB]'	Server	False	True	802.2 ns	9.220 ns	7.699 ns	804.6 ns	0.1259	0.0010	9.81 KB
'new byte[10kB]'	Workstation	False	False	399.9 ns	7.724 ns	7.225 ns	397.6 ns	2.3923	-	9.81 KB

```
// * Warnings *
MultimodalDistribution
  GcBenchmarks.'new byte[10kB]': Background Server -> It seems that the distribution is bimodal (mValue = 3.3448275862069)
```

- More settings available:
  - CpuGroups
  - AllowVeryLargeObjects
  - RetainVM
  - NoAffinitize
  - HeapAffinitizeMask
  - HeapCount

Validators



```
PS C:\Users\adsitnik\source\repos\BdnDemo\BdnDemo> dotnet run --framework netcoreapp2.1 -- --filter *
// Validating benchmarks:
Assembly BdnDemo which defines benchmarks is non-optimized
Benchmark was built without optimization enabled (most probably a DEBUG configuration). Please, build it in RELEASE.
Assembly BdnDemo which defines benchmarks is non-optimized
Benchmark was built without optimization enabled (most probably a DEBUG configuration). Please, build it in RELEASE.
Assembly BdnDemo which defines benchmarks is non-optimized
Benchmark was built without optimization enabled (most probably a DEBUG configuration). Please, build it in RELEASE.
Assembly BdnDemo which defines benchmarks is non-optimized
Benchmark was built without optimization enabled (most probably a DEBUG configuration). Please, build it in RELEASE.
Assembly BdnDemo which defines benchmarks is non-optimized
Benchmark was built without optimization enabled (most probably a DEBUG configuration). Please, build it in RELEASE.
Assembly BdnDemo which defines benchmarks is non-optimized
Benchmark was built without optimization enabled (most probably a DEBUG configuration). Please, build it in RELEASE.
Assembly BdnDemo which defines benchmarks is non-optimized
Benchmark was built without optimization enabled (most probably a DEBUG configuration). Please, build it in RELEASE.
Global total time: 00:00:00 (0.41 sec), executed benchmarks: 0
```

Compare frameworks

```
[ClrJob(baseline: true), MonoJob, CoreJob, CoreRtJob]
public class Algo_Md5VsSha256
{
    private readonly byte[] data;
    private readonly MD5 md5 = MD5.Create();
    private readonly SHA256 sha256 = SHA256.Create();

    public Algo_Md5VsSha256()
    {
        data = new byte[10000];
        new Random(42).NextBytes(data);
    }

    [Benchmark]
    public byte[] Md5() => md5.ComputeHash(data);

    [Benchmark]
    public byte[] Sha256() => sha256.ComputeHash(data);
}
```

.NET Core vs .NET vs Mono vs CoreRT

```
BenchmarkDotNet=v0.10.14.20180425-develop, OS=Windows 10.0.16299.371 (1709/FallCreatorsUpdate/Redstone3)
Intel Core i7-6700 CPU 3.40GHz (Skylake), 1 CPU, 8 logical and 4 physical cores
Frequency=3328125 Hz, Resolution=300.4695 ns, Timer=TSC
  [Host]     : .NET Framework 4.7.1 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2633.0
  Job-TOQMCM : .NET Framework 4.7.1 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2633.0
  Core       : .NET Core 2.0.6 (CoreCLR 4.6.26212.01, CoreFX 4.6.26212.01), 64bit RyuJIT
  CoreRT     : .NET CoreRT 1.0.26425.02, 64bit AOT
  Mono       : Mono 5.10.1 (Visual Studio), 64bit
```

Method	Job	Runtime	IsBaseline	Mean	Error	StdDev	Median	Scaled	ScaledSD
Md5	Default	Clr	True	21.43 us	0.4042 us	0.4655 us	21.38 us	1.00	0.00
Md5	Core	Core	Default	19.58 us	0.1170 us	0.1094 us	19.59 us	0.91	0.02
Md5	CoreRT	CoreRT	Default	19.43 us	0.1222 us	0.1084 us	19.46 us	0.91	0.02
Md5	Mono	Mono	Default	38.34 us	0.7854 us	0.9933 us	37.89 us	1.79	0.06
Sha256	Default	Clr	True	82.60 us	1.6289 us	2.6303 us	81.30 us	1.00	0.00
Sha256	Core	Core	Default	45.34 us	0.4360 us	0.3865 us	45.39 us	0.55	0.02
Sha256	CoreRT	CoreRT	Default	45.47 us	0.0616 us	0.0445 us	45.47 us	0.55	0.02
Sha256	Mono	Mono	Default	146.21 us	3.1143 us	2.9131 us	145.12 us	1.77	0.06

--runtimes

--runtimes net46 netcoreapp2.0 netcoreapp2.1

```
BenchmarkDotNet=v0.11.1.817-nightly, OS=Windows 10.0.17134.376 (1803/April2018Update/Redstone4)
Intel Core i7-5557U CPU 3.10GHz (Broadwell), 1 CPU, 4 logical and 2 physical cores
Frequency=3027349 Hz, Resolution=330.3220 ns, Timer=TSC
.NET Core SDK=2.1.403
  [Host]     : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT
  Job-LWAHYW : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.3221.0
  Job-XODHOL : .NET Core 2.0.7 (CoreCLR 4.6.26328.01, CoreFX 4.6.26403.03), 64bit RyuJIT
  Job-LCDRWL : .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT


   Method | Runtime |     Toolchain |      Mean |     Error |    StdDev |
--------- |-------- |-------------- |----------:|----------:|----------:|
 ParseInt |     Clr |         net46 | 104.55 ns | 2.6190 ns | 3.2163 ns |
 ParseInt |    Core | netcoreapp2.0 | 116.31 ns | 0.6944 ns | 0.5421 ns |
 ParseInt |    Core | netcoreapp2.1 |  98.92 ns | 0.2420 ns | 0.2146 ns |
```

Architecture: Summary

- Host process generates, builds and runs .exe per benchmark
- It helps us to get repeatable results
- It allows the users to compare different settings:
  - Legacy vs RyuJit
  - GC Workstation vs GC Server
  - .NET vs Mono vs Core vs CoreRT
- It limits us to only known frameworks
- InProcessToolchain runs in process (-i)

Diagnosers

- Plugins that allow to get some extra diagnostic information
- Can attach to the child process using available events
- Few types: extra run / no overhead / separate logic

Memory Diagnoser

- Peforms an extra iteration at the end of Actual Workload
- Uses available API:
  - `AppDomain.CurrentDomain.MonitoringTotalAllocatedMemorySize`
  - GC.GetAllocatedBytesForCurrentThread()
  - No API for Mono
- Accuracy limited to the APIs and GC allocation quantum

Memory Diagnoser sample

```
[MemoryDiagnoser]
public class AccurateAllocations
{
    [Benchmark] public void Nothing() { }
    [Benchmark] public byte[] EightBytesArray() => new byte[8];
    [Benchmark] public byte[] SixtyFourBytesArray() => new byte[64];

    [Benchmark] public Task<int> AllocateTask()
                    => Task.FromResult(default(int));
}
```

Memory Diagnoser results

```
BenchmarkDotNet=v0.11.1.817-nightly, OS=Windows 10.0.17134.376 (1803/April2018Update/Redstone4)
Intel Core i7-5557U CPU 3.10GHz (Broadwell), 1 CPU, 4 logical and 2 physical cores
Frequency=3027338 Hz, Resolution=330.3232 ns, Timer=TSC
.NET Core SDK=2.1.403
  [Host]     : .NET Core 2.0.7 (CoreCLR 4.6.26328.01, CoreFX 4.6.26403.03), 64bit RyuJIT
  DefaultJob : .NET Core 2.0.7 (CoreCLR 4.6.26328.01, CoreFX 4.6.26403.03), 64bit RyuJIT
```

Method	Mean	Error	StdDev	Gen 0/1k Op	Gen 1/1k Op	Gen 2/1k Op	Allocated Memory/Op
Nothing	0.0000 ns	0.0000 ns	0.0000 ns	-	-	-	-
EightBytesArray	3.5091 ns	0.0535 ns	0.0474 ns	0.0152	-	-	32 B
SixtyFourBytesArray	6.5717 ns	0.0996 ns	0.0831 ns	0.0419	-	-	88 B
AllocateTask	5.5788 ns	0.0365 ns	0.0324 ns	0.0343	-	-	72 B

Hardware Performance Counters

- Performs an extra run
- Uses TraceEvent, which uses ETW to get the PMCs
- Requires to run as Admin, no virtualization support
- Windows only

Hardware Counters Sample

```
[HardwareCounters(HardwareCounter.BranchMispredi
ctions, HardwareCounter.BranchInstructions)]
public class Cpu_BranchPerdictor
{

    private static int Branch(int[] data)
    {
        int sum = 0;
        for (int i = 0; i < N; i++)
            if (data[i] >= 128)
                sum += data[i];
        return sum;
    }

    private static int Branchless(int[] data)
    {
        int sum = 0;
        for (int i = 0; i < N; i++)
        {
            int t = (data[i] - 128) >> 31;
            sum += ~t & data[i];
        }
        return sum;
    }
```

```
[Benchmark]
public int SortedBranch()
        => Branch(sorted);

[Benchmark]
public int UnsortedBranch()
        => Branch(unsorted);

[Benchmark]
public int SortedBranchless()
        => Branchless(sorted);

[Benchmark]
public int UnsortedBranchless()
        => Branchless(unsorted);
```

Harware Counters Result

Method	Mean	Mispredict rate	BranchInstructions /Op	BranchMispredictions /Op
SortedBranch	21.4539 us	0,04%	70121	24
UnsortedBranch	136.1139 us	**23,70%**	68788	16301
SortedBranchless	28.6705 us	0,06%	35711	22
UnsortedBranchless	28.9336 us	0,05%	35578	17

Disassembly Diagnoser

- Attaches at the end (no extra run)
- Uses ClrMD to get the ASM, Mono.Cecil for IL
- 32 and 64 bit exe embeded in the resources
- Supports:
  - desktop .NET: LegacyJit (32 & 64 bit), RyuJIT (64 bit)
  - .NET Core 2.0+ for RyuJIT (64 & 32 bit)
  - Mono: 32 & 64 bit, **including LLVM**
  - Does not work for CoreRT **(yet)**

Disassembly Diagnoser: Sample

BdnDemo.Sum

Field .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT

```
00007ff9`2d692be0 BdnDemo.Sum.Field()
            int sum = 0;
            ^^^^^^^^^^^^
00007ff9`2d692be4 33c0            xor     eax,eax
            for (int i = 0; i < field.Length; i++)
                 ^^^^^^^^^
00007ff9`2d692be6 33d2            xor     edx,edx
            for (int i = 0; i < field.Length; i++)
                            ^^^^^^^^^^^^^^^^^
00007ff9`2d692be8 488b4908        mov     rcx,qword ptr [rcx+8]
00007ff9`2d692bec 83790800        cmp     dword ptr [rcx+8],0
00007ff9`2d692bf0 7e18            jle     00007ff9`2d692c0a
                sum += field[i];
                ^^^^^^^^^^^^^^^^
00007ff9`2d692bf2 4c8bc1          mov     r8,rcx
00007ff9`2d692bf5 413b5008        cmp     edx,dword ptr [r8+8]
00007ff9`2d692bf9 7314            jae     00007ff9`2d692c0f
00007ff9`2d692bfb 4c63ca          movsxd  r9,edx
00007ff9`2d692bfe 4303448810      add     eax,dword ptr [r8+r9*4+10h]
            for (int i = 0; i < field.Length; i++)
                                                 ^^^
00007ff9`2d692c03 ffc2            inc     edx
00007ff9`2d692c05 395108          cmp     dword ptr [rcx+8],edx
00007ff9`2d692c08 7fe8            jg      00007ff9`2d692bf2
            return sum;
            ^^^^^^^^^^^
00007ff9`2d692c0a 4883c428        add     rsp,28h
```

Local .NET Core 2.1.5 (CoreCLR 4.6.26919.02, CoreFX 4.6.26919.02), 64bit RyuJIT

```
00007ff9`2d662be0 BdnDemo.Sum.Local()
            int[] local = field;
            ^^^^^^^^^^^^^^^^^^^^
00007ff9`2d662be0 488b4108        mov     rax,qword ptr [rcx+8]
            int sum = 0;
            ^^^^^^^^^^^^
00007ff9`2d662be4 33d2            xor     edx,edx
            for (int i = 0; i < local.Length; i++)
                 ^^^^^^^^^
00007ff9`2d662be6 33c9            xor     ecx,ecx
            for (int i = 0; i < local.Length; i++)
                            ^^^^^^^^^^^^^^^^^
00007ff9`2d662be8 448b4008        mov     r8d,dword ptr [rax+8]
00007ff9`2d662bec 4585c0          test    r8d,r8d
00007ff9`2d662bef 7e0f            jle     00007ff9`2d662c00
                sum += local[i];
                ^^^^^^^^^^^^^^^^
00007ff9`2d662bf1 4c63c9          movsxd  r9,ecx
00007ff9`2d662bf4 4203548810      add     edx,dword ptr [rax+r9*4+10h]
            for (int i = 0; i < local.Length; i++)
                                                 ^^^
00007ff9`2d662bf9 ffc1            inc     ecx
00007ff9`2d662bfb 443bc1          cmp     r8d,ecx
00007ff9`2d662bfe 7ff1            jg      00007ff9`2d662bf1
            return sum;
            ^^^^^^^^^^^
00007ff9`2d662c00 8bc2            mov     eax,edx
```

Sample HTML report

BdnDemo.Sum.Field()

```
        sub     rsp,28h
        xor     eax,eax
        xor     edx,edx
        mov     rcx,qword ptr [rcx+8]
        cmp     dword ptr [rcx+8],0
        jle     M00_L01
M00_L00
        mov     r8,rcx
        cmp     edx,dword ptr [r8+8]
        jae     M00_L02
        movsxd  r9,edx
        add     eax,dword ptr [r8+r9*4+10h]
        inc     edx
        cmp     dword ptr [rcx+8],edx
        jg      M00_L00
M00_L01
        add     rsp,28h
        ret
```

ASM diffs

```
-; BenchmarkDotNet.Samples.IntroDisassemblyRyuJit.SumLocal()
-                   var local = field; // we use local variable that points to the fi
eld
-             ^^^^^^^^^^^^^^^^^^
-       mov     rax,qword ptr [rcx+8]
+; BenchmarkDotNet.Samples.IntroDisassemblyRyuJit.SumField()
                    int sum = 0;
              ^^^^^^^^^^^^
-       xor     edx,edx
-                   for (int i = 0; i < local.Length; i++)
+       xor     eax,eax
+                   for (int i = 0; i < field.Length; i++)
                  ^^^^^^^^^
-       xor     ecx,ecx
-                   for (int i = 0; i < local.Length; i++)
+       xor     edx,edx
+                   for (int i = 0; i < field.Length; i++)
                            ^^^^^^^^^^^^^^^^
-       mov     r8d,dword ptr [rax+8]
-       test    r8d,r8d
+       mov     rcx,qword ptr [rcx+8]
+       cmp     dword ptr [rcx+8],0
        jle     M00_L01
-                       sum += local[i];
+                       sum += field[i];
                 ^^^^^^^^^^^^^^^^
```

PMC + ASM

mispred branch

DisDemo.Cpu_BranchPerdictor.Branch(Int32[])

```
-        -                    int sum = 0;
                              ^^^^^^^^^^^^
-        -        00007ffc`9d2f7184 33c0            xor     eax,eax
-        -                    for (int i = 0; i < N; i++)
                                  ^^^^^^^^^
-        -        00007ffc`9d2f7186 33d2            xor     edx,edx
-        -        00007ffc`9d2f7188 4885c9          test    rcx,rcx
-        -        00007ffc`9d2f718b 742d            je      00007ffc`9d2f71ba
-        -                        if (data[i] >= 128)
                                  ^^^^^^^^^^^^^^^^^^^^
1,33%    92,81%   00007ffc`9d2f719a 4c63c2          movsxd  r8,edx
0,11%    0,00%    00007ffc`9d2f719d 468b448110      mov     r8d,dword ptr [rcx+r8*4+10h]
0,68%    0,04%    00007ffc`9d2f71a2 4181f880000000  cmp     r8d,80h
-        -        00007ffc`9d2f71a9 7c03            jl      00007ffc`9d2f71ae
-        -                            sum += data[i];
                                      ^^^^^^^^^^^^^^^
45,95%   3,62%    00007ffc`9d2f71ab 4103c0          add     eax,r8d
-        -                    for (int i = 0; i < N; i++)
                                                      ^^^
51,93%   3,52%    00007ffc`9d2f71ae ffc2            inc     edx
-        -                    for (int i = 0; i < N; i++)
                                                 ^^^^^
-        0,00%    00007ffc`9d2f71b0 81faff7f0000    cmp     edx,7FFFh
-        -        00007ffc`9d2f71b6 7ce2            jl      00007ffc`9d2f719a
-        -                    return sum;
                              ^^^^^^^^^^^
-        0,00%    00007ffc`9d2f71dd 4883c428        add     rsp,28h
```

100,00% 100,00%

Method(s) without any hardware counters:
DisDemo.Cpu_BranchPerdictor.UnsortedBranch()

PMC + ASM = skids ;(

mispred branch

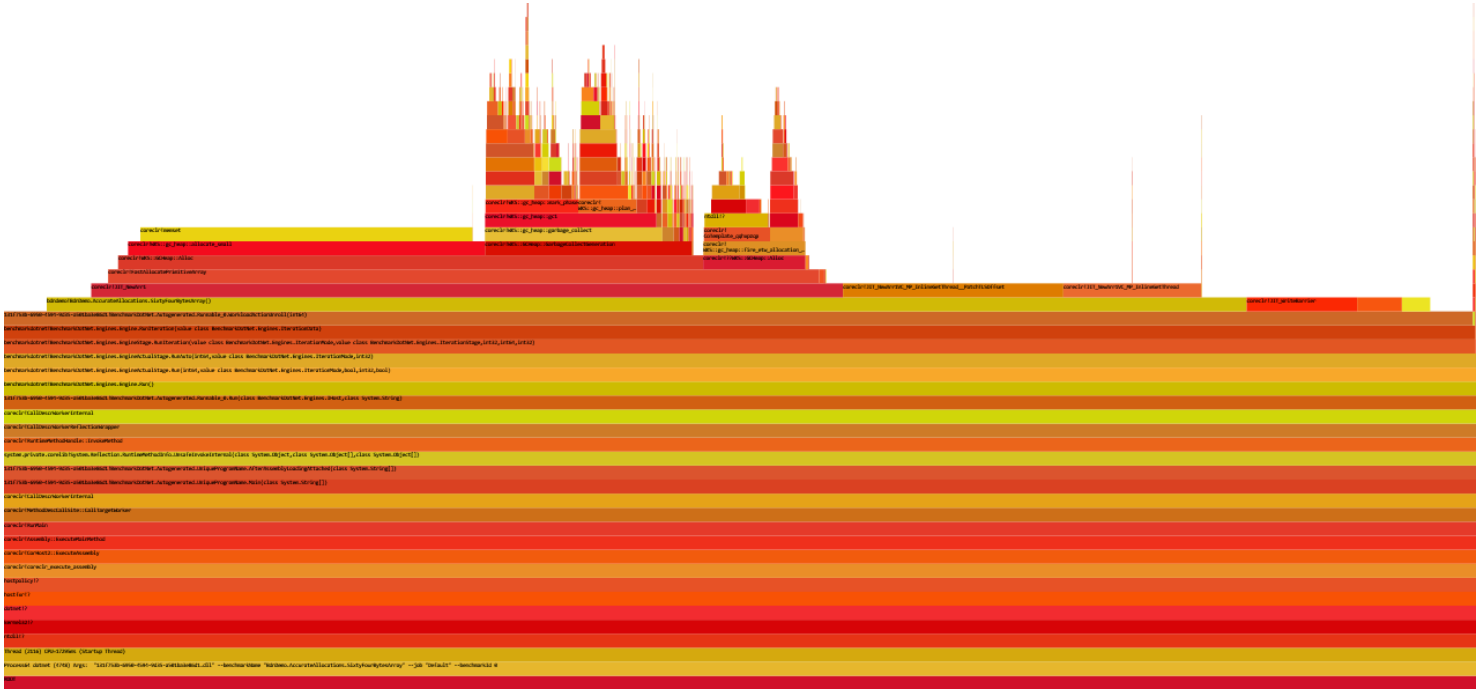
DisDemo.Cpu_BranchPerdictor.Branch(Int32[])

```
-        -                    int sum = 0;
                              ^^^^^^^^^^^^
-        -        00007ffc`9d2f7184 33c0            xor     eax,eax
-        -                    for (int i = 0; i < N; i++)
                                  ^^^^^^^^^
-        -        00007ffc`9d2f7186 33d2            xor     edx,edx
-        -        00007ffc`9d2f7188 4885c9          test    rcx,rcx
-        -        00007ffc`9d2f718b 742d            je      00007ffc`9d2f71ba
-        -                        if (data[i] >= 128)
                                  ^^^^^^^^^^^^^^^^^^^^
1,33%    92,81%   00007ffc`9d2f719a 4c63c2          movsxd  r8,edx
0,11%    0,00%    00007ffc`9d2f719d 468b448110      mov     r8d,dword ptr [rcx+r8*4+10h]
0,68%    0,04%    00007ffc`9d2f71a2 4181f880000000  cmp     r8d,80h
-        -        00007ffc`9d2f71a9 7c03            jl      00007ffc`9d2f71ae
-        -                            sum += data[i];
                                      ^^^^^^^^^^^^^^^
45,95%   3,62%    00007ffc`9d2f71ab 4103c0          add     eax,r8d
-        -                    for (int i = 0; i < N; i++)
                                                      ^^^
51,93%   3,52%    00007ffc`9d2f71ae ffc2            inc     edx
-        -                    for (int i = 0; i < N; i++)
                                              ^^^^^
-        0,00%    00007ffc`9d2f71b0 81faff7f0000    cmp     edx,7FFFh
-        -        00007ffc`9d2f71b6 7ce2            jl      00007ffc`9d2f719a
-        -                    return sum;
                              ^^^^^^^^^^^
-        0,00%    00007ffc`9d2f71dd 4883c428        add     rsp,28h
```

100,00% 100,00%

Method(s) without any hardware counters:
DisDemo.Cpu_BranchPerdictor.UnsortedBranch()

ETW Profiler



Diagnosers: Summary

- MemoryDiagnoser - accurate total size of allocated memory
- DisassemblyDiagnoser - get ASM, IL and C# for any .NET
- PmcDiagnoser – Hardware Counters on Windows
- We can combine PMC & ASM
- EtwProfiler – to profile the benchmarked code
- InliningDiagnoser uses ETW to get info about inlining
- TailCallDiagnoser uses ETW to get info about Tail Call opt
- Architecture allows for more (like integration with profilers)

BenchmarkDotNet: Summary

- Accurate, Repeatable and Stable Results
- **Powerful** Statistics
- Rich support:
  - C#, F#, VB
  - .NET 4.6+, .NET Core 2.0+, Mono, CoreRT
  - Windows, Linux, macOS
- Easy benchmark design (no boilerplate code and nice API)
- Great User Experience
- Strong community
- Very good test coverage

Do you still want to write your own harness using Stopwatch?

Questions?

Thank you!

Docs: http://benchmarkdotnet.org/

Code: https://github.com/dotnet/BenchmarkDotNet