High performance Coding with .NET Core and C#

Gergely Kalapos
Elastic

Gergely Kalapos

Web: http://kalapos.net
Twitter: @gregkalapos

- Works on observability and performance monitoring tools at Elastic
- Loves OpenSource, works in a distributed team
- Born in Hungary, lives in Austria

GitHub repo with slides and code samples

https://bit.ly/2ktZ2QZ

Agenda

- LINQ Sample - .NET Core vs. Full Framework
- Why is performance important?
- BenchmarkDotNet
- ArrayPool
- C# ref return
- Span<T>
- SIMD and Vectorization
- System.IO.Pipelines

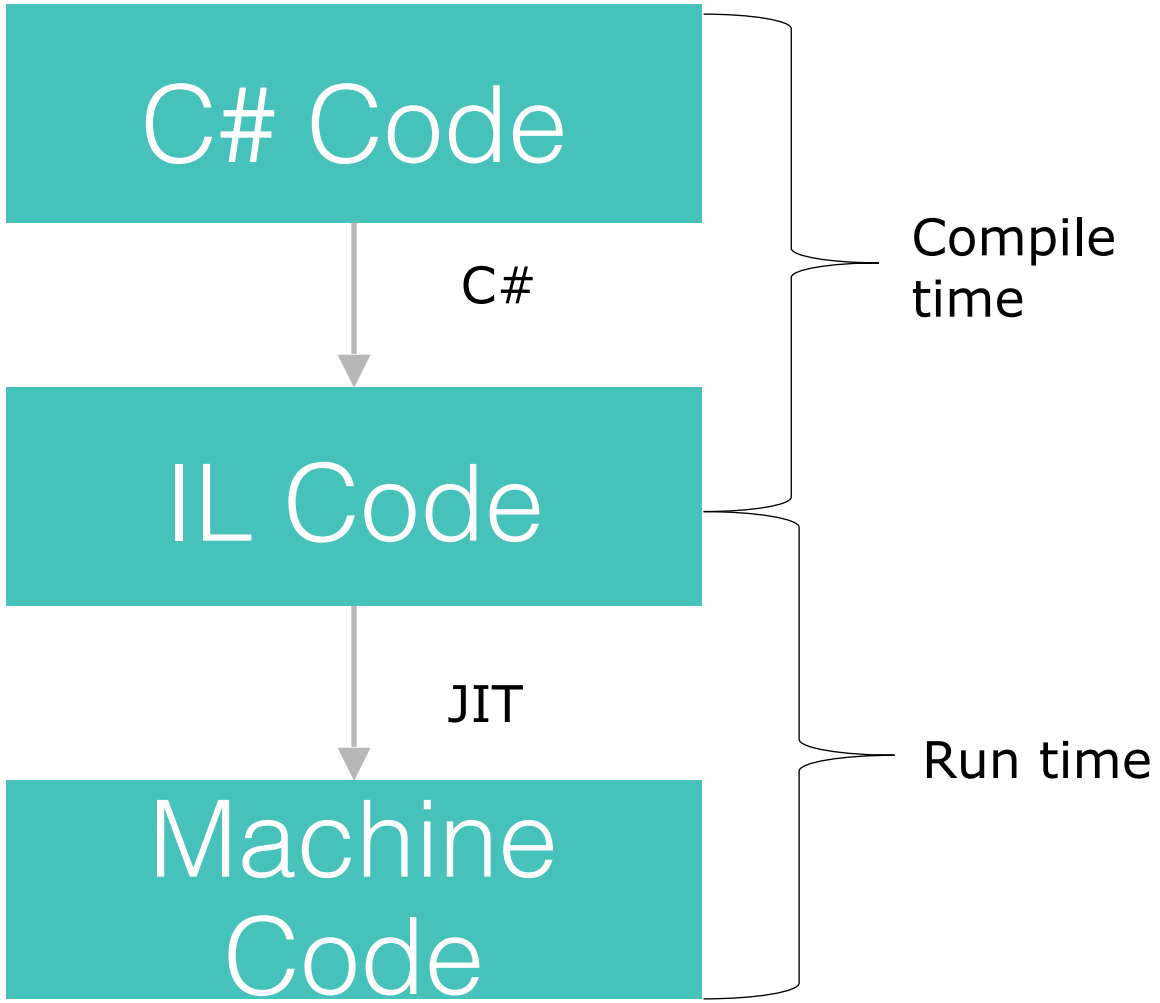
Gergely Kalapos
@gregkalapos
www.kalapos.net

Terminology

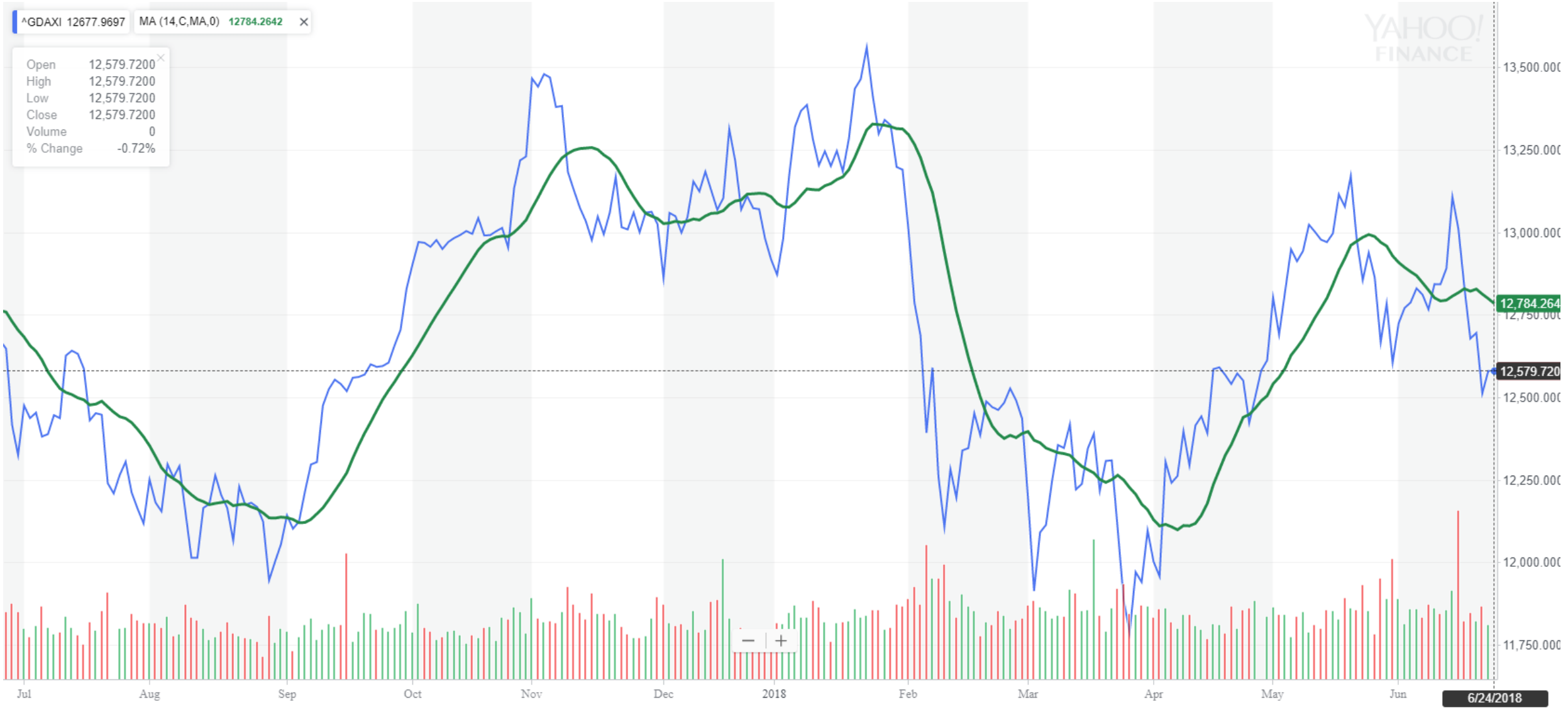
- Full Framework
- Desktop CLR

- .NET Core
- CoreCLR

- IL Code



Full Framework vs. .NET Core – LINQ - Moving average calculation



Full Framework vs .NET Core - LINQ

- Benchmarking - summary:
  - MSFT Stock
  - 14-days Simple Moving Average
  - From 1986
  - ~80.000 data points

  - https://github.com/gregkalapos/HihgPerfCsAndDotNet_Basta2019/tree/master/Sample1_Linq_FullFramework_Vs_DotNetCore

Full Framework vs .NET Core - LINQ

```
|           Method |  Job | Runtime |       Mean |     Error |    StdDev |    Gen 0 |    Gen 1 | Allocated |
|----------------- |----- |-------- |----------- |---------- |---------- |--------- |--------- |---------- |
| CalculateWithLinq |  Clr |     Clr | 541.506 ms | 2.5407 ms | 2.2523 ms | 875.0000 | 312.5000 |   3.47 MB |
| CalculateWithLinq | Core |    Core |   9.015 ms | 0.1246 ms | 0.1104 ms | 281.2500 |  93.7500 |   1.14 MB |

// * Hints *
Outliers
  Program.CalculateWithLinq: Clr  -> 1 outlier  was  removed
  Program.CalculateWithLinq: Core -> 1 outlier  was  removed

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen 0     : GC Generation 0 collects per 1k Operations
  Gen 1     : GC Generation 1 collects per 1k Operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ms      : 1 Millisecond (0.001 sec)
```

~60x Performance improvement thanks to Community Contribution (**Jon Hanna**)

https://github.com/dotnet/corefx/commit/a087c2d97d818762c1b4dd2c6305a2d9e836631c#diff-5c3753c526c75a2790d71c348c0c364aR246

https://github.com/dotnet/corefx/blob/master/src/System.Linq/src/System/Linq/Skip.cs#L35

BenchmarkDotNet

- Open-source micro-benchmarking tool for .NET
- Add NuGet package: “BenchmarkDotNet”
- https://www.nuget.org/packages/BenchmarkDotNet/

BenchmarkDotNet - why? Why not Stopwatch?

- JIT Compilation
- 32 vs. 64 bit
- Full Framework vs. NET Core vs. Xamarin
- RyuJIT vs. Classic JIT

Covering all these environments is very hard!

BenchmarkDotNet

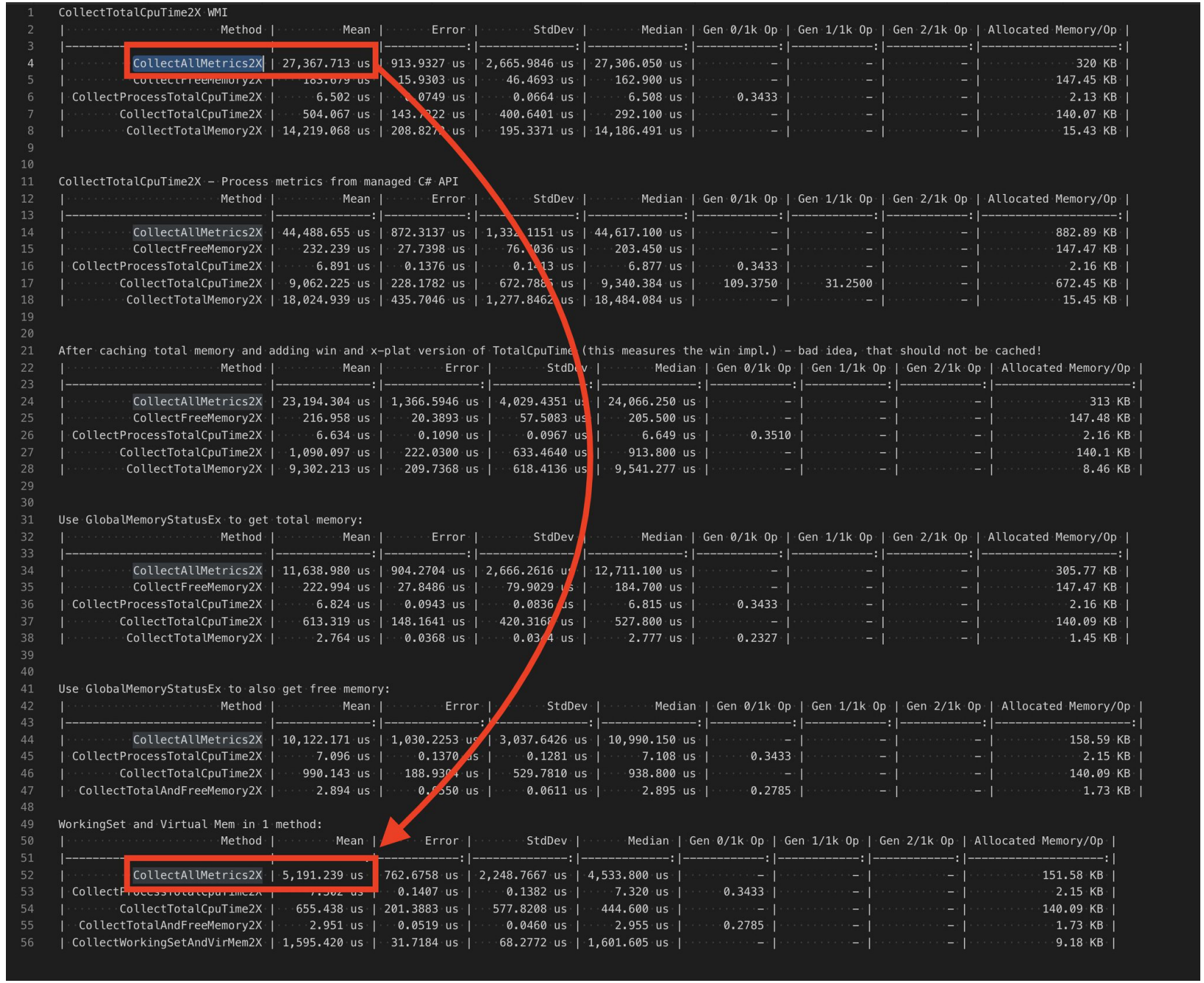
- Warm-up phase for every benchmark
- Multiple iterations
- Nice reports
- Shows the generated IL and machine code
- Easy setup for different GC Settings, JIT Compliers, .NET Flavors

BenchmarkDotNet

```
namespace BenchmarkSample
{
    [MemoryDiagnoser]
    [ClrJob, CoreJob]
    public class Program
    {
        static void Main(string[] args)
        {
            var summary = BenchmarkRunner.Run<Program>();
        }
        [Benchmark]
        public void BenchmarkMethod()
        {
            //work to measure
        }
    }
```

BenchmarkDotNet - output

```
|           Method |  Job | Runtime |       Mean |     Error |    StdDev |    Gen 0 |    Gen 1 | Allocated |
|----------------- |----- |-------- |-----------:|----------:|----------:|---------:|---------:|----------:|
| CalculateWithLinq |  Clr |     Clr | 541.506 ms | 2.5407 ms | 2.2523 ms | 875.0000 | 312.5000 |   3.47 MB |
| CalculateWithLinq | Core |    Core |   9.015 ms | 0.1246 ms | 0.1104 ms | 281.2500 |  93.7500 |   1.14 MB |

// * Hints *
Outliers
  Program.CalculateWithLinq: Clr  -> 1 outlier  was  removed
  Program.CalculateWithLinq: Core -> 1 outlier  was  removed

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen 0     : GC Generation 0 collects per 1k Operations
  Gen 1     : GC Generation 1 collects per 1k Operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ms      : 1 Millisecond (0.001 sec)
```



CollectTotalCpuTime2X WMI

Method	Mean	Error	StdDev	Median	Gen 0/1k Op	Gen 1/1k Op	Gen 2/1k Op	Allocated Memory/Op
CollectAllMetrics2X	27,367.713 us	913.9327 us	2,665.9846 us	27,306.050 us	-	-	-	320 KB
CollectFreeMemory2X	183.679 us	15.9303 us	46.4693 us	162.900 us	-	-	-	147.45 KB
CollectProcessTotalCpuTime2X	6.502 us	0.0749 us	0.0664 us	6.508 us	0.3433	-	-	2.13 KB
CollectTotalCpuTime2X	504.067 us	143.7222 us	400.6401 us	292.100 us	-	-	-	140.07 KB
CollectTotalMemory2X	14,219.068 us	208.8277 us	195.3371 us	14,186.491 us	-	-	-	15.43 KB

CollectTotalCpuTime2X - Process metrics from managed C# API

Method	Mean	Error	StdDev	Median	Gen 0/1k Op	Gen 1/1k Op	Gen 2/1k Op	Allocated Memory/Op
CollectAllMetrics2X	44,488.655 us	872.3137 us	1,332.1151 us	44,617.100 us	-	-	-	882.89 KB
CollectFreeMemory2X	232.239 us	27.7398 us	76.7036 us	203.450 us	-	-	-	147.47 KB
CollectProcessTotalCpuTime2X	6.891 us	0.1376 us	0.1413 us	6.877 us	0.3433	-	-	2.16 KB
CollectTotalCpuTime2X	9,062.225 us	228.1782 us	672.7885 us	9,340.384 us	109.3750	31.2500	-	672.45 KB
CollectTotalMemory2X	18,024.939 us	435.7046 us	1,277.8462 us	18,484.084 us	-	-	-	15.45 KB

After caching total memory and adding win and x-plat version of TotalCpuTime (this measures the win impl.) - bad idea, that should not be cached!

Method	Mean	Error	StdDev	Median	Gen 0/1k Op	Gen 1/1k Op	Gen 2/1k Op	Allocated Memory/Op
CollectAllMetrics2X	23,194.304 us	1,366.5946 us	4,029.4351 us	24,066.250 us	-	-	-	313 KB
CollectFreeMemory2X	216.958 us	20.3893 us	57.5083 us	205.500 us	-	-	-	147.48 KB
CollectProcessTotalCpuTime2X	6.634 us	0.1090 us	0.0967 us	6.649 us	0.3510	-	-	2.16 KB
CollectTotalCpuTime2X	1,090.097 us	222.0300 us	633.4640 us	913.800 us	-	-	-	140.1 KB
CollectTotalMemory2X	9,302.213 us	209.7368 us	618.4136 us	9,541.277 us	-	-	-	8.46 KB

Use GlobalMemoryStatusEx to get total memory:

Method	Mean	Error	StdDev	Median	Gen 0/1k Op	Gen 1/1k Op	Gen 2/1k Op	Allocated Memory/Op
CollectAllMetrics2X	11,638.980 us	904.2704 us	2,666.2616 us	12,711.100 us	-	-	-	305.77 KB
CollectFreeMemory2X	222.994 us	27.8486 us	79.9029 us	184.700 us	-	-	-	147.47 KB
CollectProcessTotalCpuTime2X	6.824 us	0.0943 us	0.0836 us	6.815 us	0.3433	-	-	2.16 KB
CollectTotalCpuTime2X	613.319 us	148.1641 us	420.3168 us	527.800 us	-	-	-	140.09 KB
CollectTotalMemory2X	2.764 us	0.0368 us	0.0344 us	2.777 us	0.2327	-	-	1.45 KB

Use GlobalMemoryStatusEx to also get free memory:

Method	Mean	Error	StdDev	Median	Gen 0/1k Op	Gen 1/1k Op	Gen 2/1k Op	Allocated Memory/Op
CollectAllMetrics2X	10,122.171 us	1,030.2253 us	3,037.6426 us	10,990.150 us	-	-	-	158.59 KB
CollectProcessTotalCpuTime2X	7.096 us	0.1370 us	0.1281 us	7.108 us	0.3433	-	-	2.15 KB
CollectTotalCpuTime2X	990.143 us	188.9304 us	529.7810 us	938.800 us	-	-	-	140.09 KB
CollectTotalAndFreeMemory2X	2.894 us	0.0550 us	0.0611 us	2.895 us	0.2785	-	-	1.73 KB

WorkingSet and Virtual Mem in 1 method:

Method	Mean	Error	StdDev	Median	Gen 0/1k Op	Gen 1/1k Op	Gen 2/1k Op	Allocated Memory/Op
CollectAllMetrics2X	5,191.239 us	762.6758 us	2,248.7667 us	4,533.800 us	-	-	-	151.58 KB
CollectProcessTotalCpuTime2X	7.302 us	0.1407 us	0.1382 us	7.320 us	0.3433	-	-	2.15 KB
CollectTotalCpuTime2X	655.438 us	201.3883 us	577.8208 us	444.600 us	-	-	-	140.09 KB
CollectTotalAndFreeMemory2X	2.951 us	0.0519 us	0.0460 us	2.955 us	0.2785	-	-	1.73 KB
CollectWorkingSetAndVirMem2X	1,595.420 us	31.7184 us	68.2772 us	1,601.605 us	-	-	-	9.18 KB

Why is Performance important? – Cloud

- Azure App Service costs

Basic:

INSTANZ	KERNE	RAM	STORAGE	PREISE
B1	1	1,75 GB	10 GB	~$54,75/Monat
B2	2	3,50 GB	10 GB	~$109,50/Monat
B3	4	7 GB	10 GB	~$219/Monat

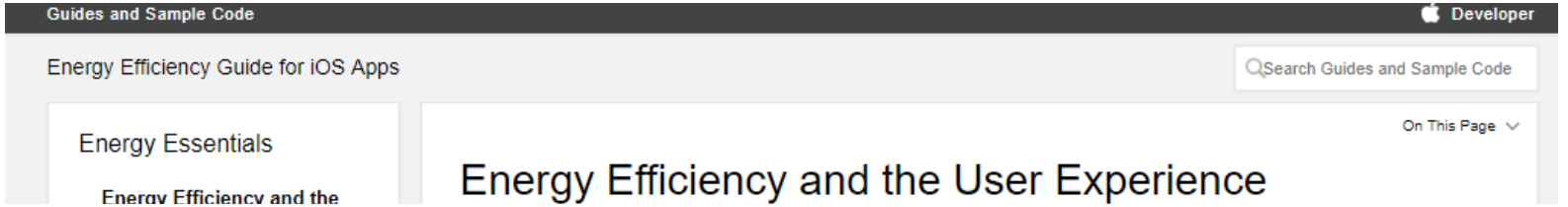
In case an application runs on a:

- Single B1 instance: 54,75$/Month
- Single P3v2 instance: 584$/Month
- 50 P3v2 instances : 29200$/Month

Premium:

INSTANZ	KERNE	RAM	STORAGE	PREISE
P1v2	1	3,50 GB	250 GB	~$146/Monat
P2v2	2	7 GB	250 GB	~$292/Monat
P3v2	4	14 GB	250 GB	~$584/Monat

Why is Performance important? – Mobile



Your Obligation as a Developer

Even small inefficiencies in apps add up, significantly affecting battery life, performance, and responsiveness. As an app developer, you have an obligation to make sure your app runs as efficiently as possible. Use recommended APIs so the system can make smart decisions about how best to manage your app and the resources it uses. Whenever possible, batch and reduce network operations, and avoid unnecessary updates to the user interface. Power-intensive operations should be under the user's control. If a user is playing a graphics-heavy game, for example, the user should not be surprised if the activity consumes power. Strive to make your app absolutely idle when it is not responding to user input.

By adhering to recommended guidelines, you can make big contributions to the overall energy efficiency of the platform and the satisfaction of your users.

Techempower Benchmark

Plaintext

Best (bar chart) | Data table | Latency | Framework overhead

Best plaintext responses per second, Dell R440 Xeon Gold + 10 GbE (352 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	hyper	7,007,513	100.0%	0	Mcr	Rus	Rus	Hyp	Lin	Rea
2	tokio-minihttp	7,006,181	100.0%	0	Mcr	Rus	Rus	tok	Lin	Rea
3	ulib-plaintext_fit	7,004,608	100.0%	2	Plt	C++	Non	ULi	Lin	Rea
4	actix	7,000,911	99.9%	0	Mcr	Rus	Non	act	Lin	Rea
5	ulib	6,998,172	99.9%	3	Plt	C++	Non	ULi	Lin	Rea
6	libreactor	6,997,422	99.9%	0	Mcr	C	Non	Non	Lin	Rea
7	actix-raw	6,996,104	99.8%	0	Plt	Rus	Non	act	Lin	Rea
8	atreugo-prefork	6,995,436	99.8%	0	Plt	Go	Non	Non	Lin	Rea
9	firenio_http_lite	6,994,344	99.8%	0	Plt	Jav	fir	Non	Lin	Rea
10	aspcore	6,993,704	99.8%	0	Plt	C#	.NE	kes	Lin	Rea
11	aspcore-rntx	6,990,400	99.8%	0	Plt	C#	.NE	kes	Lin	Rea
12	wizzardo-http	6,987,612	99.7%	0	Mcr	Jav	Non	Non	Lin	Rea
13	fasthttp	6,983,911	99.7%	0	Plt	Go	Non	Non	Lin	Rea
14	may-minihttp	6,977,134	99.6%	0	Mcr	Rus	Rus	may	Lin	Rea
15	fasthttp-prefork	6,969,608	99.5%	0	Plt	Go	Non	Non	Lin	Rea
16	rapidoid-http-fast	6,968,492	99.4%	0	Plt	Jav	Rap	Non	Lin	Rea
17	rapidoid	6,936,304	99.0%	0	Plt	Jav	Rap	Non	Lin	Rea
18	aspcore-corert	6,853,345	97.8%	0	Plt	C#	.NE	kes	Lin	Rea
19	thruster	6,537,703	93.3%	0	Mcr	Rus	Rus	Non	Lin	Rea
20	cpoll_cppsp	6,505,764	92.8%	0	Plt	C++	Non	Non	Lin	Rea
21	atreugo	6,476,674	92.4%	0	Plt	Go	Non	Non	Lin	Rea

Source: techempower.com

Techempower Benchmark

77	jooby2-jetty	1,272,306	18.2%	0	Ful	Jav	Jty	Non	Lin	Rea
78	aspcore-mvc	1,255,837	17.9%	0	Ful	C#	.NE	kes	Lin	Rea
79	amber	1,232,049	17.6%	0	Ful	Cry	Non	Non	Lin	Rea
80	facil.io	1,221,373	17.4%	0	Mcr	C	Non	Non	Lin	Rea
81	vibed-ldc	1,168,920	16.7%	0	Plt	D	Non	Non	Lin	Rea
82	vibed-ldc-pgsql	1,148,524	16.4%	0	Plt	D	Non	Non	Lin	Rea
83	workerman	1,125,572	16.1%	0	Plt	PHP	Non	wor	Lin	Rea
84	ratpack	1,107,946	15.8%	0	Mcr	Jav	Nty	Non	Lin	Rea
85	finatra	1,053,298	15.0%	0	Mcr	Sca	Nty	Non	Lin	Rea
86	swift-nio	1,010,416	14.4%	0	Plt	Swi	Non	Non	Lin	Rea
87	nodejs	997,327	14.2%	0	Plt	JS	njs	Non	Lin	Rea
88	nodejs-chakra	994,225	14.2%	0	Plt	JS	njs	Non	Lin	Rea
89	finch	993,490	14.2%	0	Mcr	Sca	Nty	Non	Lin	Rea
90	servlet3-sync	987,384	14.1%	0	Plt	Jav	Svt	tom	Lin	Rea
91	finagle	975,009	13.9%	0	Mcr	Sca	Nty	Non	Lin	Rea
92	kemal	967,739	13.8%	0	Ful	Cry	Non	Non	Lin	Rea
93	akka-http	938,771	13.4%	0	Mcr	Sca	Akk	Non	Lin	Rea
94	gramework	920,600	13.1%	0	Plt	Go	Non	Non	Lin	Rea
95	go-pgx-prefork	913,917	13.0%	0	Plt	Go	Non	Non	Lin	Rea

Source: techempower.com

Techempower Benchmark

319	sailsjs	7,737	0.1%	0	Ful	JS	njs	Non	Lin	Rea
320	spyne-nginx-uwsgi	7,718	0.1%	0	Mcr	Py	Non	ngx	Lin	Rea
321	bottle-nginx-uwsgi	7,607	0.1%	0	Mcr	Py	uWS	ngx	Lin	Rea
322	flask-nginx-uwsgi	7,420	0.1%	0	Mcr	Py	Non	ngx	Lin	Rea
323	laravel	7,215	0.1%	0	Ful	PHP	Non	ngx	Lin	Rea
324	akka-http-slick-postgres	4,309	0.1%	0	Mcr	Sca	Akk	Non	Lin	Rea
325	ubiquity-react	3,977	0.1%	7	Ful	PHP	Non	php	Lin	Rea
326	hapi-nginx	3,779	0.1%	0	Mcr	JS	Non	Non	Lin	Rea
327	officefloor-rapidoid	2,725	0.0%	0	Ful	Jav	off	woo	Lin	Rea
329	aspnet-mono-ngx	1,862	0.0%	34,451	Plt	C#	.NE	ngx	Lin	Rea
330	webware	1,604	0.0%	0	Mcr	Py	Non	Non	Lin	Rea
331	hunt-dmd	1,092	0.0%	16,380	Plt	D	Non	Non	Lin	Rea
332	hunt	1,088	0.0%	16,330	Plt	D	Non	Non	Lin	Rea
333	beetlex	1,066	0.0%	0	Mcr	C#	.NE	bee	Lin	Rea
334	cherrypy-py3	900	0.0%	0	Mcr	Py	Non	Non	Lin	Rea
335	sanic	808	0.0%	24,248	Mcr	Py	Non	Non	Lin	Rea

Source: techempower.com

.NET Blog - Performance Improvements in .NET Core

- Post with perf. improvements with benchmarks

https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-core-3-0/

https://blogs.msdn.microsoft.com/dotnet/2017/06/07/performance-improvements-in-net-core/

Performance Improvements in .NET Core 3.0

May 15th, 2019

Back when we were getting ready to ship .NET Core 2.0, I wrote a blog post exploring some of the many performance improvements that had gone into it. I enjoyed putting it together so much and received such a positive response to the post that I did it again for .NET Core 2.1, a version for which performance was also a significant focus. With //build last week and .NET Core 3.0's release now on the horizon, I'm thrilled to have an opportunity to do it again.

.NET Core 3.0 has a ton to offer, from Windows Forms and WPF, to single-file executables, to async enumerables, to platform intrinsics, to HTTP/2, to fast JSON reading and writing, to assembly unloadability, to enhanced cryptography, and on and on and on... there is a wealth of new functionality to get excited about. For me, however, performance is the primary feature that makes me excited to go to work in the morning, and there's a staggering amount of performance goodness

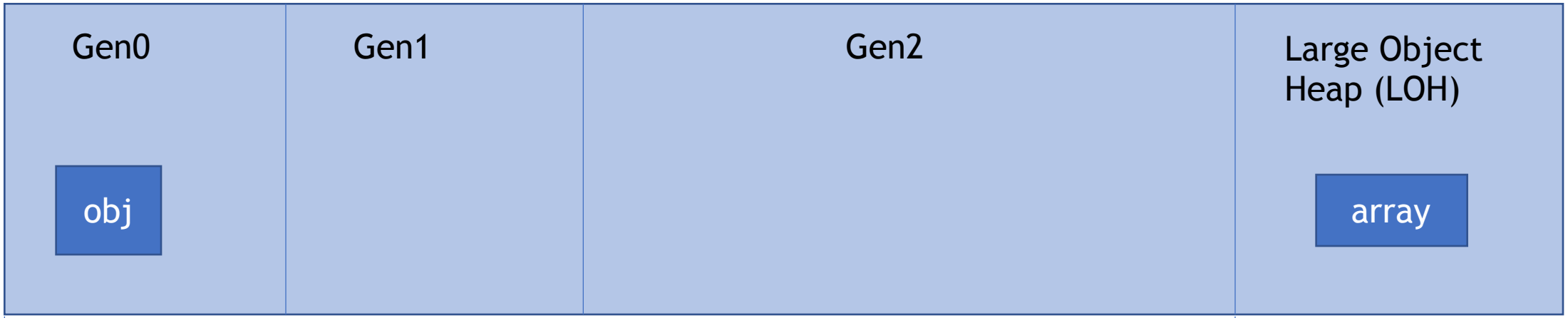
ArrayPool

Gergely Kalapos
@gregkalapos
www.kalapos.net

Managed Heap - GC Segments

```
var obj = new Object();
```

```
var array = new int[100_000];
```



Small Object Heap (SOH)

Objects bigger than 85.000 Bytes (*GCLOHThreshold)

Large Object Heap Collection = Full GC (Gen0+Gen1+Gen2+LOH)

Arrays and GC

- Method with array allocation

```
public void MethodCalledOften()
{
        var myArray = new int[15];
        //use myArray...
}
```

- myArray is allocation on the heap -> more work for the GC
- In each call memory is allocated -> GC cleans up -> and in the next round memory is allocated again

Arrays and GC

- Method with a bigger array

```
public void MethodCalledOften()
{
        var myArray = new int[256*1024];
        //use myArray...
}
```

- myArray is allocated on the Large Object Heap -> even more work for the GC

Solution: ArrayPool<T>

- Holds a number of objects ready to use
- .NET Standard 2.1
- NuGet package: System.Buffers - can be used already with .NET Core 1.0, Included in .NET Core 2.1 SDK - Microsoft.NETCore.App
- **ArrayPool<T>.Shared.Rent(int length)**: gives you an array
- **ArrayPool<T>.Shared.Return(T[] array)**: you give the array back to the pool
- https://github.com/dotnet/corefx/blob/master/src/Common/src/CoreLib/System/Buffers/ArrayPool.cs

ArrayPool

- Instead of allocating an array each time, use ArrayPool:

```
public void MethodCalledOften()
{
    var arr = ArrayPool<int>.Shared.Rent(256 * 1024);
    Console.WriteLine(arr[0].ToString());
    try
    {
        //use arr.
    }
    finally
    {
        ArrayPool<int>.Shared.Return(arr);
    }
}
```

Benchmark Result

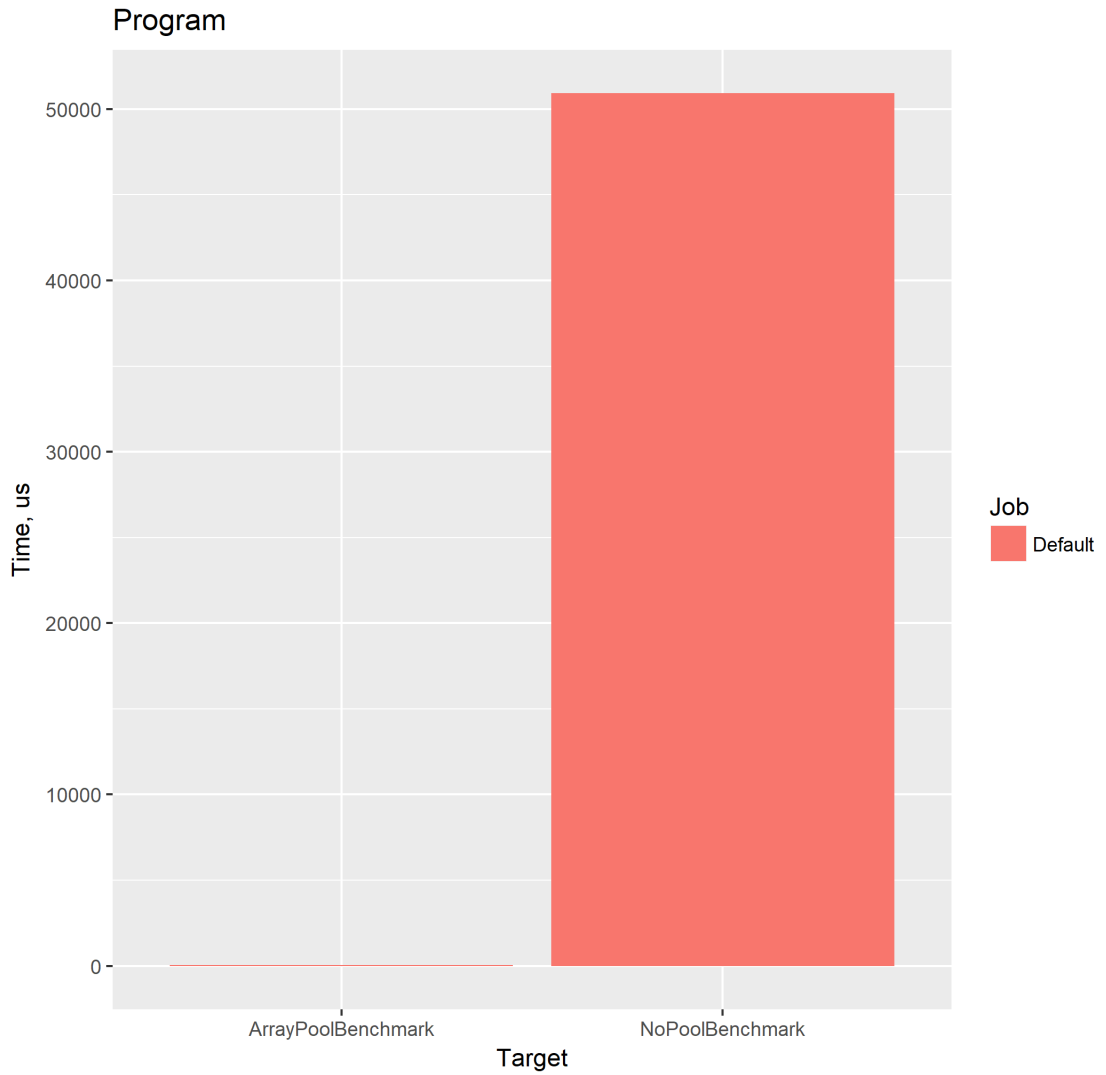
1000 iterations: ArrayPool.Shared.Rent() vs. new Array[]

https://github.com/g

naster/ArrayPoolSample

C:\Windows\System32\cmd.exe

```
Total time: 00:00:52 (52.94

// * Summary *

BenchmarkDotNet=v0.10.14, OS
Intel Core i5-5200U CPU 2.20
.NET Core SDK=2.1.300
  [Host]     : .NET Core 2.1
  DefaultJob : .NET Core 2.1

             Method |
------------------- -------
 ArrayPoolBenchmark      50
    NoPoolBenchmark  50,912
```




```
 1 |       Gen 2 |    Allocated |
--:|------------:|-------------:|
 - |           - |          0 B |
00 | 333312.5000 | 1048600000 B |
```

BenchmarkDotNet v0.10.14

ArrayPool – A few more things

- ArrayPool<T>.Shared is the "Shared Pool”
- Max array size in Shared Pool: 2^{20} (1024*1024)
- Calling Shared.Rent(n) with n > 2^{20}: new array will be allocated -> no pooling
- ArrayPool<T>.Create(maxArrayLength, maxArraysPerBucket) creates a new pool. With that you can create pools with arrays bigger than 2^{20}

ArrayPool – Creating a custom pool

- Pooling Arrays with size > 2^{20} (1024 * 1024)

```
public ArrayPool<int> MyPool =
          ArrayPool<int>.Create(1024*1024*3, 10);

public void MethodWithCustomPool()
{
    var arr = MyPool.Rent(1024*1024*2);
    //use arr...
    MyPool.Return(arr);
}
```

ArrayPool – Summary

- Very simple API -> goal: avoid GC (especially Gen2 GC)
- Shared Property returns the Shared-Pool (max. 2^{20})
- The Create method creates a new Pool
- Where is ArrayPool used? E.g.: Kestrel, ASP.NET Core, System.IO.Pipelines

Ref return, ref local

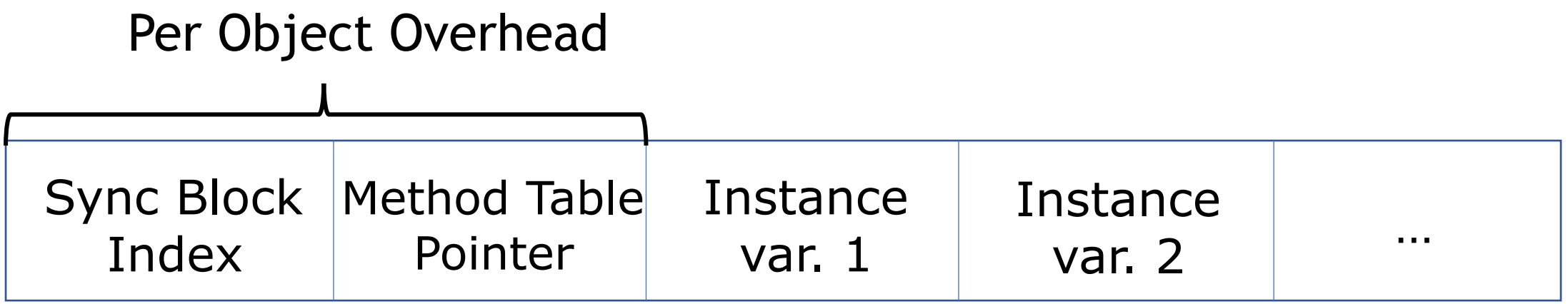
Gergely Kalapos
@gregkalapos
www.kalapos.net

Reference types – object header

- Advantage: Polymorphism, Inheritance, lock(){}
- Disadvantage: higher memory usage, can be only allocated on the heap

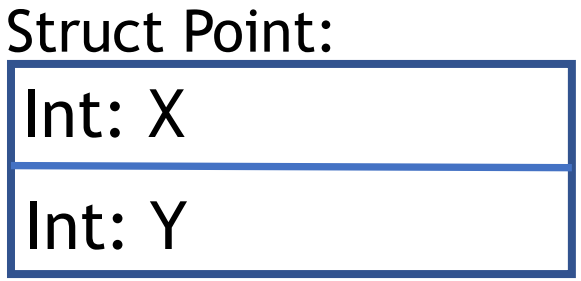
```
lock(myOb)
{
  // . . .
}
```

```
class Point
{
   int x, y;
}
```



Value Types on .NET (and .NET Core)

- Contains the value directly
- E.g.: Numerical Typen, Bool, Char, Date, Structs, Enums
- With “=“ is the contained value copied
- With “==“ the contained values are compared




```
Point p = new Point();
```

Benchmark: class vs. struct

```
public class
Point_Class
{
  public int x;
  public int y;
}
```

VS.

```
public struct
Point_Struct
{
  public int x;
  public int y;
}
```

Benchmark: class vs. struct

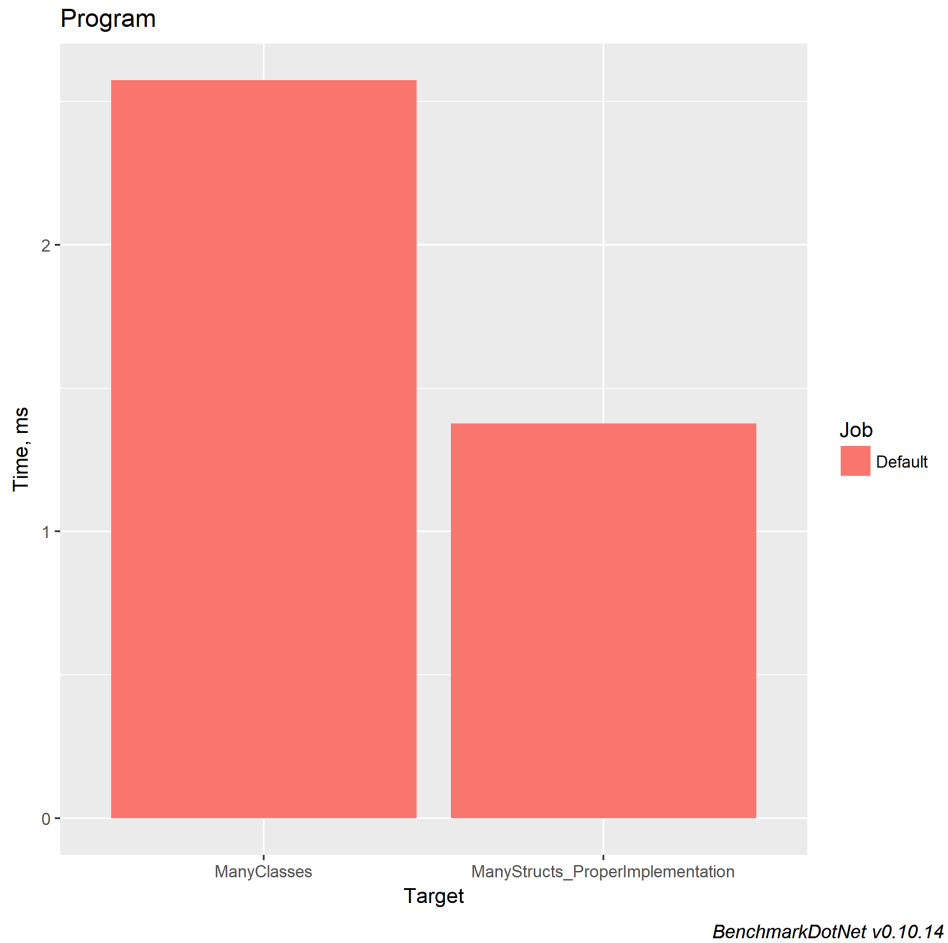
```
static List<Point_Class> ClassList
 = new List<Point_Class>();

[Benchmark]
public static void ManyClasses()
{
  for (int i = 0; i < 1_000_000; i++)
  {
    if(ClassList[i].x == -1)
    {
      Debug.WriteLine("-1");
    }
  }
}
```

```
static List<Point_Struct> StructList
  = new List<Point_Struct>();

[Benchmark]
public static void ManyStructs()
{
  for (int i = 0; i < 1_000_000; i++)
  {
    if (StructList[i].x == -1)
    {
       Debug.WriteLine("-1");
    }
  }
}
```

Benchmark: class vs. struct

https://github.com/gregkalapos/HihgPerfCsAndDotNet_Basta2019/tree/master/RefAndSpan/StructVsClass

```
BenchmarkDotNet=v0.10.14, OS=Windows 10.0.17134
Intel Core i5-5200U CPU 2.20GHz (Broadwell), 1 CPU, 4 logical and 2 phys
.NET Core SDK=2.1.300
  [Host]     : .NET Core 2.1.0 (CoreCLR 4.6.26515.07, CoreFX 4.6.26515.0
  DefaultJob : .NET Core 2.1.0 (CoreCLR 4.6.26515.07, CoreFX 4.6.26515.0


                          Method |     Mean |     Error |    StdDev |
-------------------------------- |---------:|----------:|----------:|
                      ManyClasses | 2.574 ms | 0.0094 ms | 0.0088 ms |
 ManyStructs_ProperImplementation | 1.376 ms | 0.0044 ms | 0.0041 ms |

// * Legends *
  Mean   : Arithmetic mean of all measurements
  Error  : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  1 ms   : 1 Millisecond (0.001 sec)

// ***** BenchmarkRunner: End *****
// * Artifacts cleanup *
```



Value Types - Disadvantages

- No OOP, no *lock(valuetype){}*
- Value Semantic: By default value types are copied (the value inside it is copied)

```
public struct Matrix_Struct
{
      public int i00, i01, i02;
      public int i10, i11, i12;
      public int i20, i21, i22;
}

public static void DoSomethingWithMatrix(Matrix_Struct
matrix_Struct)
{
      //...
}
```

Value Types - Benchmark

```
public class Matrix_Class
{
  public int i00, i01, i02;
  public int i10, i11, i12;
  public int i20, i21, i22;
}
```

```
[Benchmark]
public static void BenchmarkMatrixClass()
{
  PrintFirstItem(MatrixClass);
}

public static void
PrintFirstItem(Matrix_Class matrix_Class)
=> Debug.WriteLine(matrix_Class.i00);
```

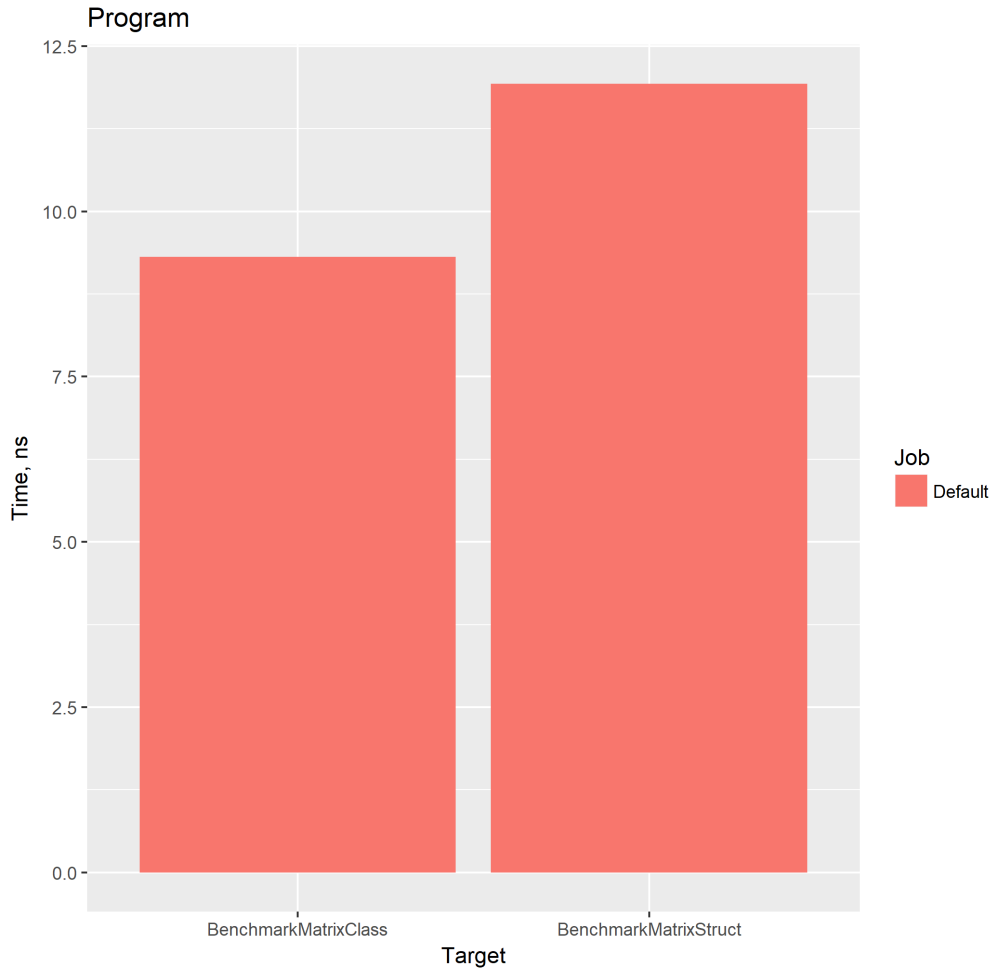
```
public struct Matrix_Struct
{
 public int i00, i01, i02;
 public int i10, i11, i12;
 public int i20, i21, i22;
}
```

```
[Benchmark]
public static void
BenchmarkMatrixStruct()
{
  PrintFirstItem(MatrixStruct);
}

public static void
PrintFirstItem(Matrix_Struct
matrix_Struct)
=> Debug.WriteLine(matrix_Struct.i00);
```

Value Types - Benchmark

- Class vs. Struct as Method Parameter

```
C:\Windows\System32\cmd.exe
.NET Core SDK=2.1.300
  [Host]     : .NET Core 2.1.0 (CoreCLR 4.6.26515.07, CoreFX
  DefaultJob : .NET Core 2.1.0 (CoreCLR 4.6.26515.07, CoreFX


               Method |      Mean |     Error |    StdDev |
--------------------- |----------:|----------:|----------:|
  BenchmarkMatrixClass |  9.281 ns | 0.0512 ns | 0.0479 ns |
 BenchmarkMatrixStruct | 12.248 ns | 0.0131 ns | 0.0116 ns |
```



https://github.com/gregkalapos/HihgPerfCsAndDotNet_Basta2019/tree/master/RefAndSpan/StructVsClass_MethodParameter

Value Types

- Solution: pass by reference

```
public struct Matrix_Struct
{
        public int i00, i01, i02;
        public int i10, i11, i12;
        public int i20, i21, i22;
}
```

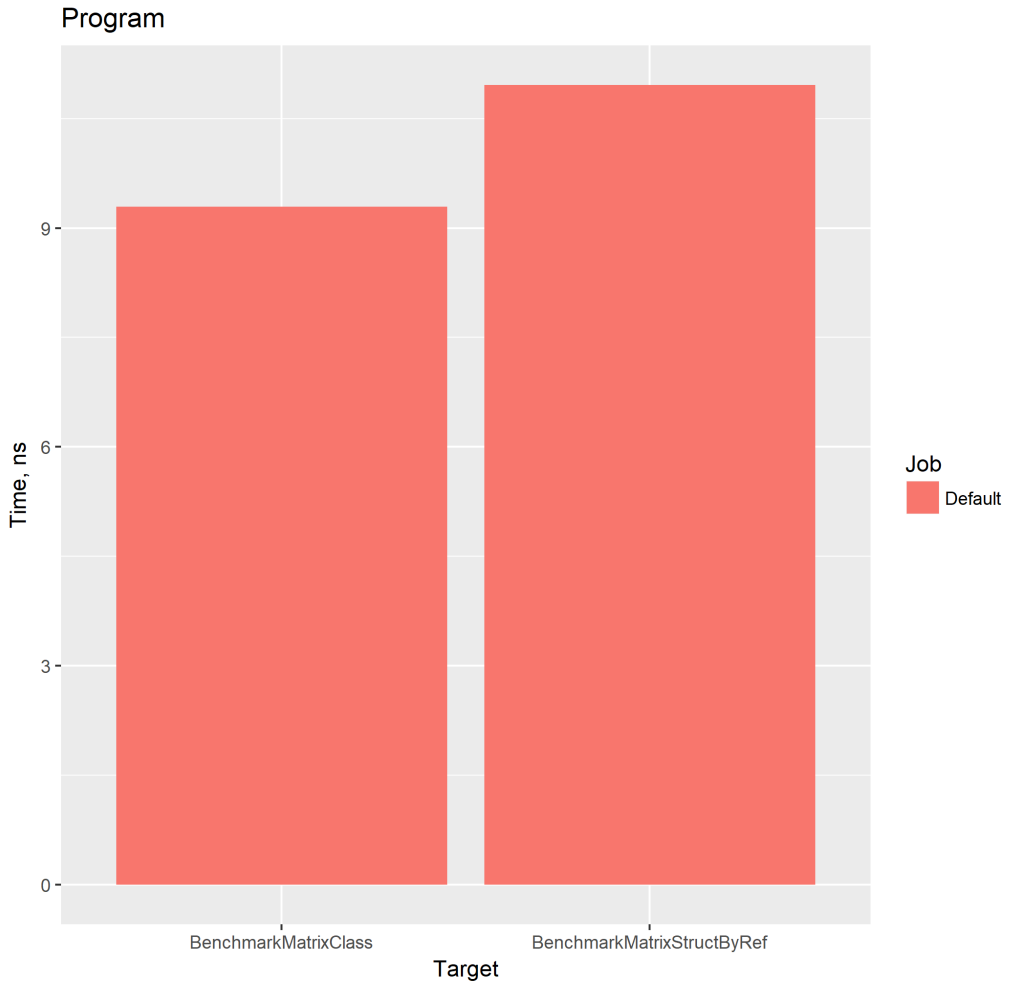
```
public static void DoSomethingWithMatrix(ref Matrix_Struct matrix_Struct)
{
        //...
}
```

Value Types

- Solution: pass by reference

C:\Windows\System32\cmd.exe

Method	Mean	Error	StdDev
BenchmarkMatrixClass	9.358 ns	0.0789 ns	0.0659 ns
BenchmarkMatrixStructByRef	10.912 ns	0.0637 ns	0.0564 ns



https://github.com/gregkalapos/HihgPerfCsAndDotNet_Basta2019/tree/master/RefAndSpan/StructVsClass_MethodParameter_PassByReference

Ref keyword in C# = Managed pointer in the CLR

C#:

```
public static void PrintFirstItem_Struct(Matrix_Struct matrix_Struct)
```

IL:

```
.method public hidebysig static
        void PrintFirstItem_Struct (
            valuetype SampleProject.Matrix_Struct matrix_Struct
        ) cil managed
```

C#:

```
public static void PrintFirstItem_Struct_Ref(ref Matrix_Struct matrix_Struct)
```

IL:

```
.method public hidebysig static
        void PrintFirstItem_Struct_Ref (
            valuetype  SampleProject.Matrix_Struct& matrix_Struct
        ) cil managed
```

Ref return

- Since C# 7 we can also use the “ref” keyword with the type

```
public static ref Matrix DoSomethingWithMatrix(ref Matrix
matrix)
{
        //
        return ref matrix;
}
```

```
.method public hidebysig static
    valuetype SampleApp.Matrix& DoSomethingWithMatrix (
        valuetype  SampleApp.Matrix& matrix
    ) cil managed
```

Ref return – Sample 1

```
public static ref Matrix GetBiggerByReference(ref Matrix matrix1, ref Matrix
matrix2)
{
  var sum1 = matrix1.i00 + matrix1.i01 + matrix1.i02 +
    matrix1.i10 + matrix1.i11 + matrix1.i12 +
    matrix1.i20 + matrix1.i21 + matrix1.i22;

  var sum2 = matrix2.i00 + matrix2.i01 + matrix2.i02 +
    matrix2.i10 + matrix2.i11 + matrix2.i12 +
    matrix2.i20 + matrix2.i21 + matrix2.i22;

  if (sum2 > sum1)
    return ref matrix2;
  else
    return ref matrix1;
}
```

Ref return – Benchmark

```
public static ref Matrix GetBiggerByReference(ref Matrix matrix1, ref Matrix
matrix2)
```

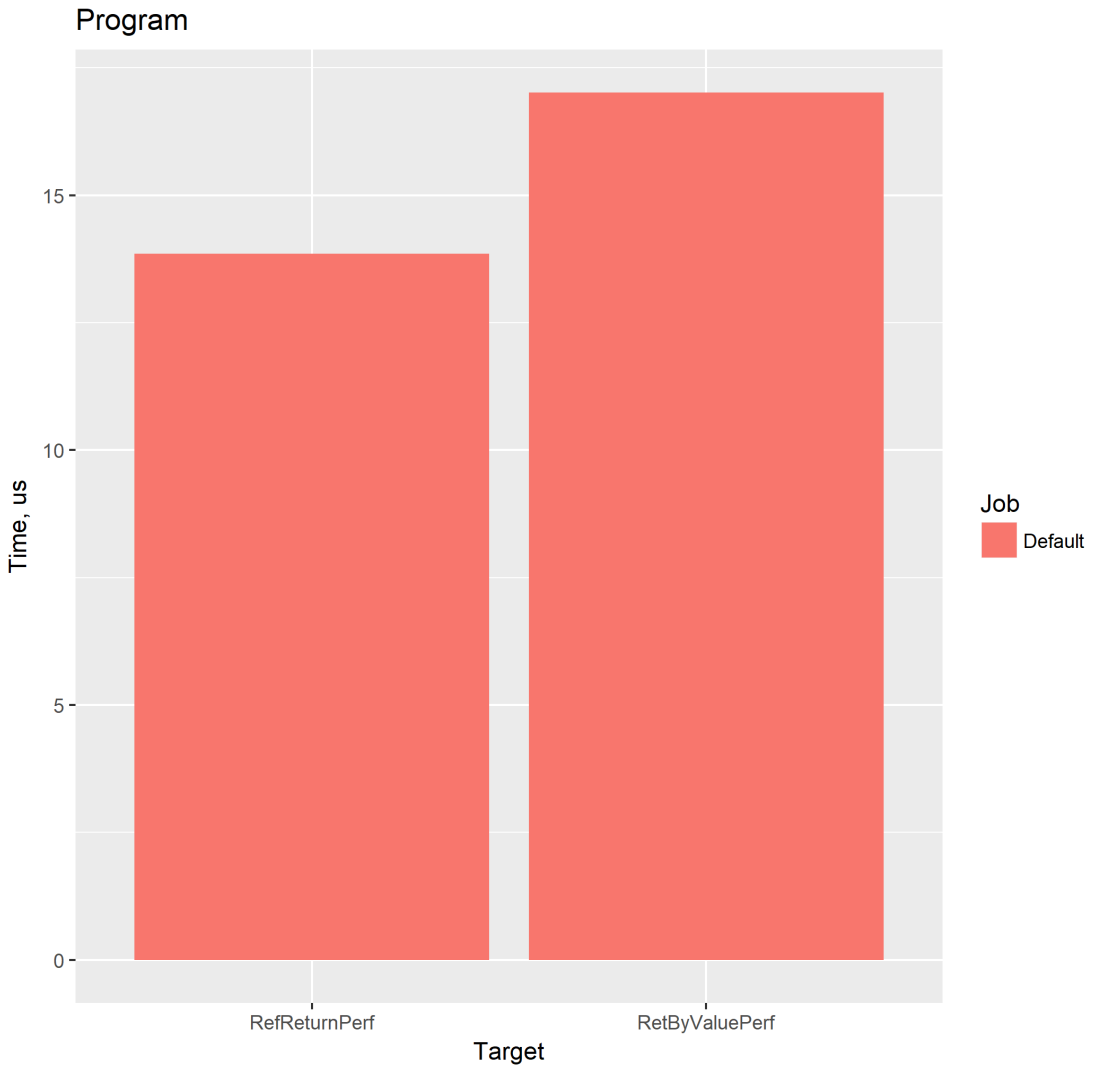
Vs.

```
public static Matrix GetBiggerByValue(Matrix matrix1, Matrix matrix2)
```

```
[Benchmark]
public static void RefReturnPerf()
{
   for (int i = 0; i < 1000; i++)
   {
     var item1 = matrixList[i].Item1;
     var item2 = matrixList[i].Item2;

     ref Matrix biggerMatrix = ref GetBiggerByReference(ref item1, ref item2);
     Debug.WriteLine(biggerMatrix.i00);
   }
}
```

Ref return - Benchmark

Method	Mean	Error	StdDev
RefReturnPerf	13.85 us	0.0724 us	0.0677 us
RetByValuePerf	17.01 us	0.0852 us	0.0797 us



https://github.com/gregkalapos/HihgPerfCsAndDotNet_Basta2019/tree/master/RefAndSpan/RefReturnMatrix

BenchmarkDotNet v0.10.14

Ref return – Sample 2

```
class MyList
{
  private int[] _items = { 3, 4, 7, 1, 4, 6 };
  public ref int GetRefToSmallestItem()
  {
     var smallest = _items[0];
     int index = 0;
     for (int i = 1; i < _items.Length; i++)
     {
       if (_items[i] < smallest)
       {
         smallest = _items[i];
         index = i;
       }
     }
     return ref _items[index];
   }

}
```

Ref return – Sample 2

```
static void Main(string[] args)
{
        MyList l = new MyList();
        ref int i = ref l.GetRefToSmallestItem();
        i = 42;
}
```

```
.method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{    .maxstack 2
    .entrypoint
    .locals init (
        [0] class CollectionWithRef.MyLis
t,
        [1] int32&
    )
```

```
    /* 0x000002E6 0A          */ IL_0006: stloc.0
    /* 0x000002E7 06          */ IL_0007: ldloc.0
    /* 0x000002E8 6F01000006  */ IL_0008:
callvirt  instance int32& CollectionWithRef.MyList::GetRefToSmallestIte
m()
    /* 0x000002ED 0B          */ IL_000D: stloc.1
    /* 0x000002EE 07          */ IL_000E: ldloc.1
    /* 0x000002EF 1F2A        */ IL_000F: ldc.i4.s  42
```

Ref in C# 7 – what can we return with “ref"?

- Ref Parameter

```
public ref int MethodReturningIntByRef(ref int myInt)
{
      //...do something with ‘myInt’
      return ref myInt;
}
```

- Out Parameter

```
public ref int MethodReturningIntByRef(out int myInt)
{
      myInt = 42;
      //...do something with 'myInt’
      return ref myInt;
}
```

Ref in C# 7 – what can we return with “ref"?

- In Parameter (C# 7.2)

```
public ref readonly int MethodReturningIntByRef(in int
myInt)
{
      //...do something with 'myInt’
      return ref myInt;
}
```

- Instance variables of a type

```
class MyClass
{
      private int myInt;

      public ref int GetRefToMyInt() => ref myInt;
}
```

Ref in C# 7 – what CAN'T we return with "ref"

- Lokale variables

```
public ref int MyMethod()
{
      int i = 42;
      return ref i;
}
```

- null

```
public ref object MyMethod()
{
      return ref null;
}
```

- this

```
public ref object MyMethod()
{
      return ref this;
}
```

Ref in C# 7 – what CAN'T we return with "ref"

- async methods

```
public async ref Task<int> MyMethod()
{
      //let's do something async:
      await Task.Delay(500);
      return ref 42;
}
```

- consts

```
const int I = 42;
public ref int MyMethod()
{
      return ref I;
}
```

- enum

```
public enum Day { Monday, Friday }
public ref Day MyMethod()
{
      var day = Day.Monday;
      return ref day;
}
```

Ref return – Where is it used?

- ASP.NET Core
- Span<T>

Span<T>

Gergely Kalapos
@gregkalapos
www.kalapos.net

Allocation in .NET

- Managed memory

```
byte[] myArray = new byte[100];
```

- Unmanaged memory

```
IntPtr myArray =
Marshal.AllocHGlobal(100);
//...use myArray
Marshal.FreeHGlobal(myArray);
```

- Stack memory

```
byte* myArray = stackalloc byte[100];
```

Accepting handles to different types of memory in an API

- “High Performance" libraries usually use managed and unmanaged (e.g. stack) memory as well (E.g.: ASP.NET Core/Kestrel)
- But it’s very hard to deal with this

```
public void ProcessByteArray(byte[] array)
public void ProcessByteArray(void* array)

public void ProcessByteArray(byte[] array, int offset)
public void ProcessByteArray(void* array, int offset)
```

Span<T>

- Represents a contiguous region of memory
- An abstraction over every type of memory that we can allocate in .NET
- .NET Standard 2.1

```
public void ProcessByteArray(Span<byte> array)
```

Span<T> - wrapping memory from the managed heap

```
byte[] myArray = new byte[100];
Span<byte> myArraySpan = new Span<byte>(myArray);
ProcessByteArray(myArraySpan);
```

Span<T> - wrapping unmanaged memory

```
IntPtr myArray = Marshal.AllocHGlobal(100);
Span<byte> nativeSpan;
unsafe
{
     nativeSpan = new Span<byte>(myArray.ToPointer(), 100);
}
ProcessByteArray(nativeSpan);

Marshal.FreeHGlobal(myArray);
```

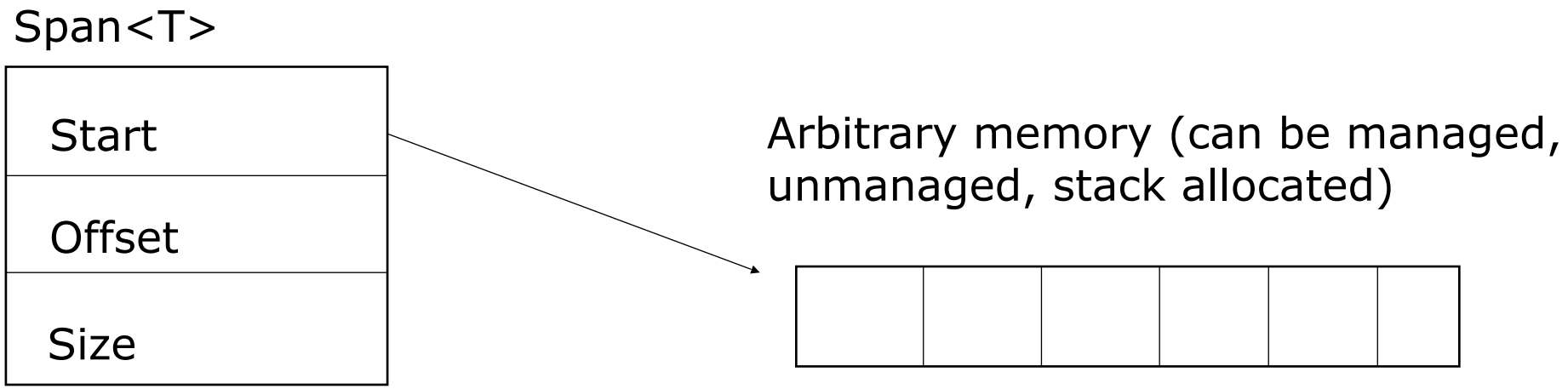
Span<T> - wrapping stack memory

Since C# 7.2:

```
Span<byte> stackSpan;
unsafe
{
    byte* myArray = stackalloc byte[100];
    stackSpan = new Span<byte>(myArray, 100);
}
ProcessByteArray(stackSpan);
```

```
Span<byte> stackSpan = stackalloc byte[100];
ProcessByteArray(stackSpan);
```

Span<T>


Span<T>
Start
Offset
Size
Arbitrary memory (can be managed, unmanaged, stack allocated)

Span<T> API

- Span constructors:

```
public Span(T[] array);

public Span(void* pointer, int length);

public Span(T[] array, int start, int length)
```

- Indexing:

```
public ref T this[int index] { get; }
```

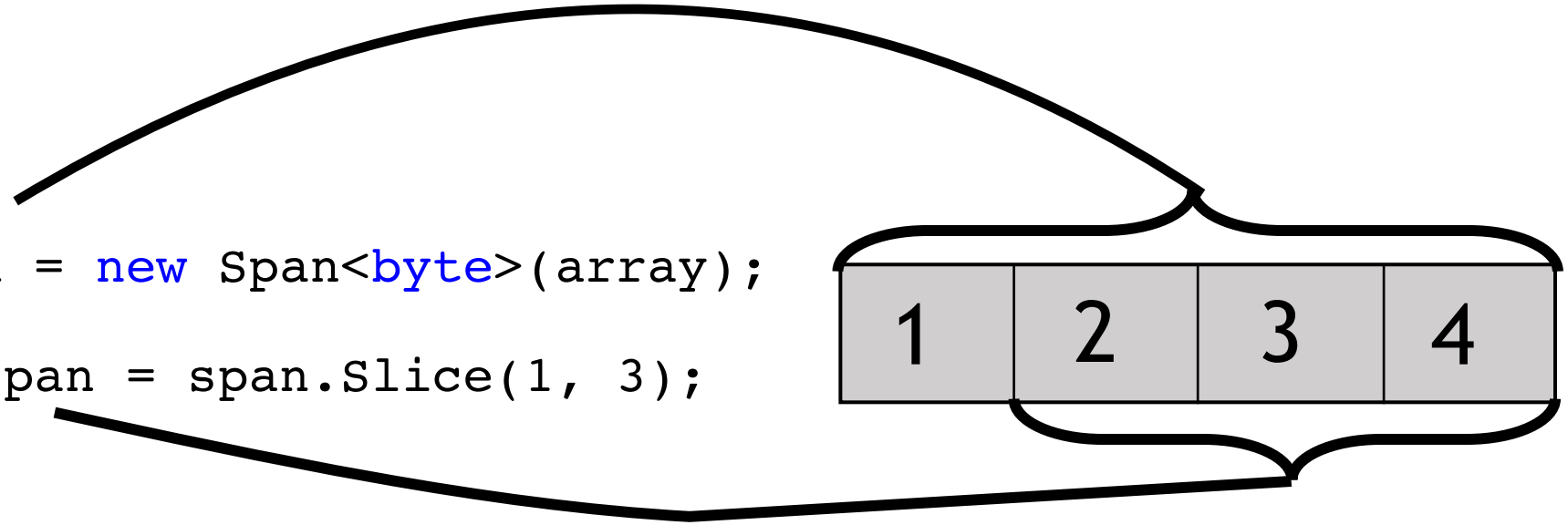
- Slicing:

```
public Span<T> Slice(int start);

public Span<T> Slice(int start, int length);
```

(*Not a complete list of all methods!)

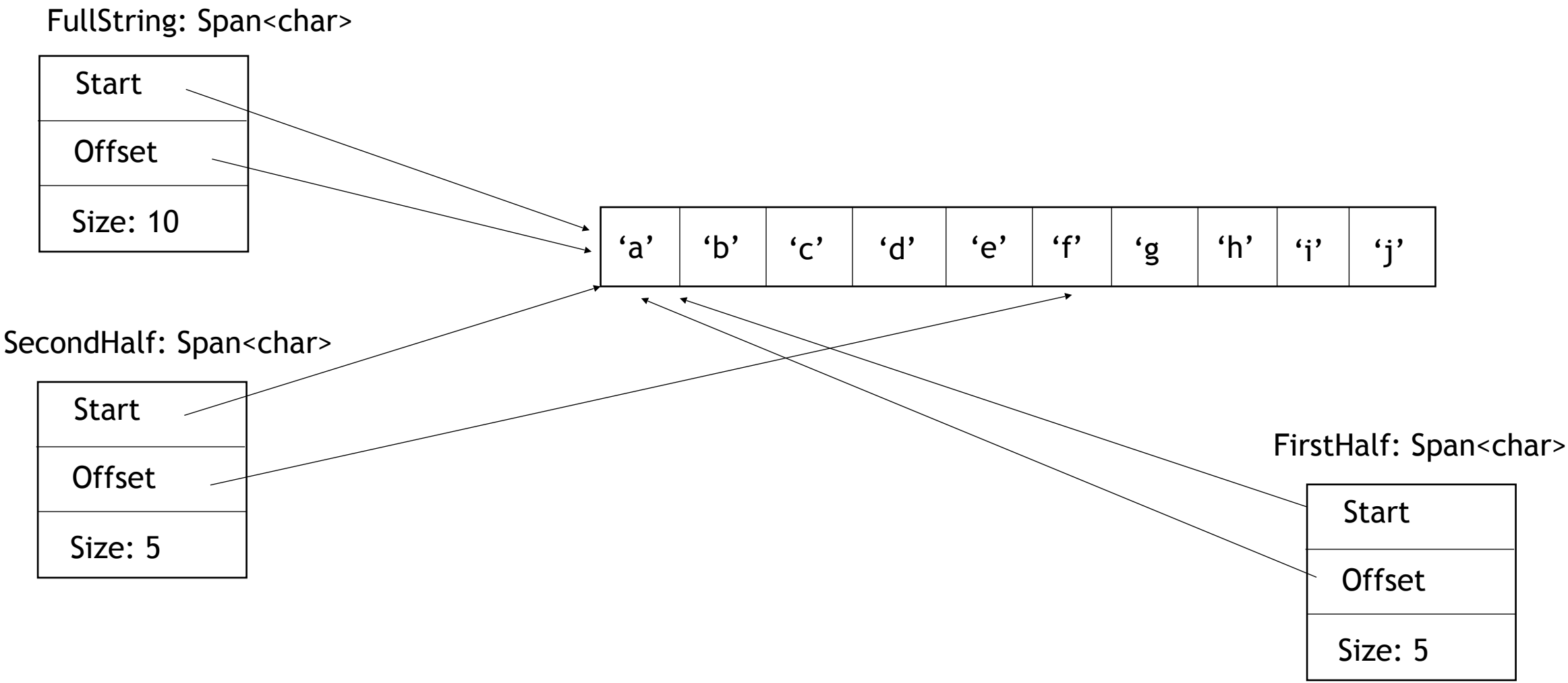
Span<T>.Slice(start, length)

```
byte[] array = new byte[] { 1, 2, 3, 4 };

Span<byte> span = new Span<byte>(array);

Span<byte> newSpan = span.Slice(1, 3);
```



This does not allocate at all

Span<T> - Performance Advantage



Span<T> - String parsing

```
public static void WorkWithSpan()
{
    ReadOnlySpan<char> str = @"Request URL: http://kalapos.net/
        Request Method:GET
        Status Code:200 OK
        Remote Address:81.10.250.159:80
        Referrer Policy:no-referrer-when-downgrade
        Accept:texvt/html,application/xhtml+xml,application/xml;q=0.9,
        Accept-Encoding:gzip, deflate
        Accept-Language:de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4,hu;q=0.2
        Connection:keep-alive
        Cookie:__cfduid=d37ec605870fbb75a1b5b9616bf161ed01507890791;
           _ga=GA1.2.460922426.1476689834; _gid=GA1.2.1163677089.1508317942
        Host:kalapos.net
        Upgrade-Insecure-Requests:1
        User-Agent:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like              Gecko) Chrome/61.0.3163.100 Safari/537.36".AsSpan();

        ParseStringSpan(str);
}
```

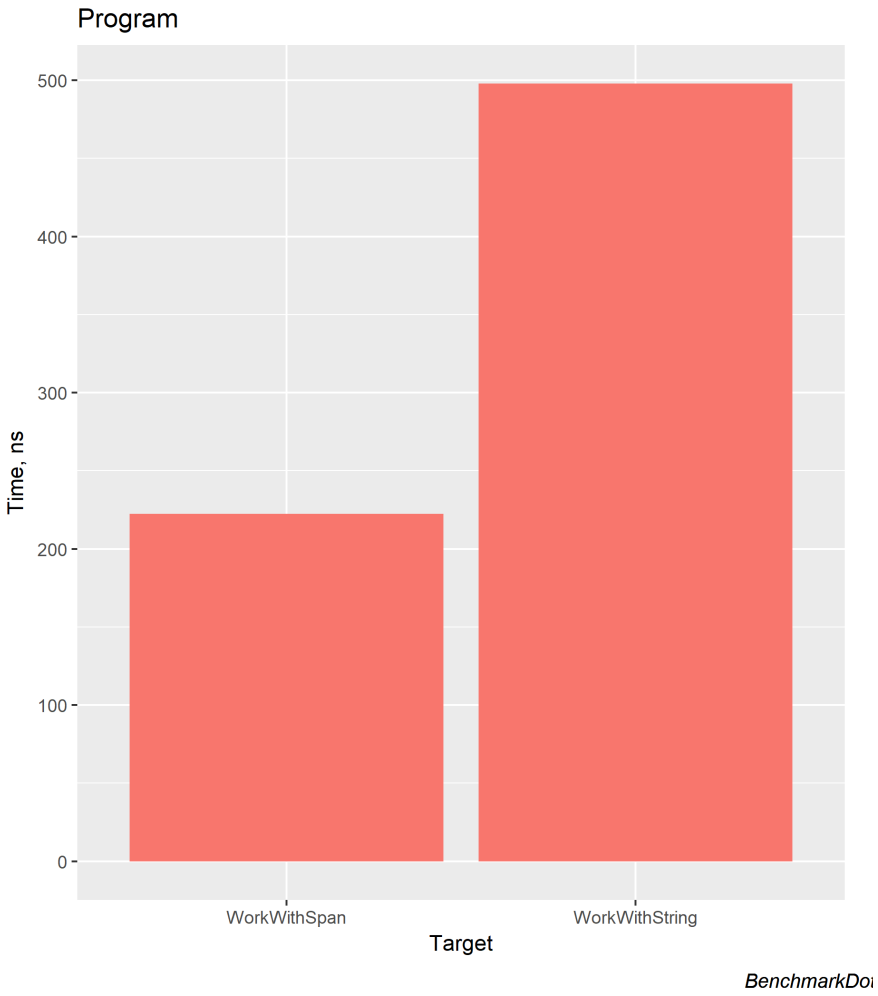
Span<T> - String parsing

```
public static void
ParseStringSpan(ReadOnlySpan<char> str)
{
 var url = str.Slice(13, 19);
 var method = str.Slice(58, 3);
 Int32.TryParse(str.Slice(84, 3), out int
statusCode);
 var remoteAddr = str.Slice(116, 14);
 var referrerPolicy = str.Slice(157, 26);
 var accept = str.Slice(201, 85);
 var acceptEncoding = str.Slice(313, 13);
 var acceptLanguage = str.Slice(353, 44);
 var connection = str.Slice(419, 10);
 var cookie = str.Slice(447, 118);
 var host = str.Slice(581, 11);
 Int32.TryParse(str.Slice(629, 1), out
int  updgradeInsecureRequest);
 var userAgent = str.Slice(652, 115);
}
```

```
public static void ParseStringClassic(String
str)
{
 var url = str.Substring(13, 19);
 var method = str.Substring(58, 3);
 Int32.TryParse(str.Substring(84, 3), out int
statusCode);
 var remoteAddr = str.Substring(116, 14);
 var referrerPolicy = str.Substring(157, 26);
 var accept = str.Substring(201, 85);
 var acceptEncoding = str.Substring(313, 13);
 var acceptLanguage = str.Substring(353, 44);
 var connection = str.Substring(419, 10);
 var cookie = str.Substring(447, 118);
 var host = str.Substring(581, 11);
 Int32.TryParse(str.Substring(629, 1), out int
updgradeInsecureRequest);
 var userAgent = str.Substring(652, 115);
}
```

Span<T> - Benchmark

```
C:\Windows\System32\cmd.exe
// * Summary *

BenchmarkDotNet=v0.10.14, OS=Windows 10.0.17134
Intel Core i5-5200U CPU 2.20GHz (Broadwell), 1 CPU, 4 logical and 2 physic
.NET Core SDK=2.1.300
  [Host]     : .NET Core 2.1.0 (CoreCLR 4.6.26515.07, CoreFX 4.6.26515.06)
  DefaultJob : .NET Core 2.1.0 (CoreCLR 4.6.26515.07, CoreFX 4.6.26515.06)


         Method |     Mean |     Error |    StdDev |  Gen 0 | Allocated |
--------------- |---------:|----------:|----------:|-------:|----------:|
 WorkWithString | 509.3 ns | 1.8835 ns | 1.7619 ns | 0.8183 |    1288 B |
   WorkWithSpan | 222.5 ns | 0.6024 ns | 0.5340 ns |      - |       0 B |

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen 0     : GC Generation 0 collects per 1k Operations
  Allocated : Allocated memory per single operation (managed only, inclusi
  1 ns      : 1 Nanosecond (0.000000001 sec)
```



https://github.com/gregkalapos/HihgPerfCsAndDotNet_Basta2019/tree/master/RefAndSpan/StringParsingWithSpan

Span<T> use cases

- Parsing
- System.Text.Json APIs
- ASP.NET Core

Memory<T>

- Span<T> is a ref struct - it can only live on the stack

```
public async Task<Span<int>> MethodWithSpan()
{
        await Task.Delay(4);
        var span = new Span<int>(new int[]{ 1, 2, 3 });
        return span;
}
```

```
public Span<int> MethodWithSpan2()
{
        Span<int> span = stackalloc int[5];
        return span;
}
```

- Memory<T>: Same functionality as Span<T>, but can be on the heap

.NET Standard 2.1

- ~3K new API
- Lots of them are Span<T> or ReadonlySpan<T> overloads

```
public readonly struct Int32 : IComparable, IComparable<int>, IConvertible, IEquatable<int>, IFormattable {
  public const int MaxValue = 2147483647;
  public const int MinValue = -2147483648;
  public Int32 CompareTo(Int32 value);
  public Int32 CompareTo(object value);
  public bool Equals(Int32 obj);
  public override bool Equals(object obj);
  public override Int32 GetHashCode();
  public TypeCode GetTypeCode();
+ public static Int32 Parse(ReadOnlySpan<char> s, NumberStyles style = NumberStyles.Integer, IFormatProvider provider = null);
  public static Int32 Parse(string s);
  public static Int32 Parse(string s, NumberStyles style);
  public static Int32 Parse(string s, NumberStyles style, IFormatProvider provider);
  public static Int32 Parse(string s, IFormatProvider provider);
  bool System.IConvertible.ToBoolean(IFormatProvider provider);
  byte System.IConvertible.ToByte(IFormatProvider provider);
  char System.IConvertible.ToChar(IFormatProvider provider);
  DateTime System.IConvertible.ToDateTime(IFormatProvider provider);
  decimal System.IConvertible.ToDecimal(IFormatProvider provider);
  double System.IConvertible.ToDouble(IFormatProvider provider);
  short System.IConvertible.ToInt16(IFormatProvider provider);
  Int32 System.IConvertible.ToInt32(IFormatProvider provider);
  long System.IConvertible.ToInt64(IFormatProvider provider);
  sbyte System.IConvertible.ToSByte(IFormatProvider provider);
  float System.IConvertible.ToSingle(IFormatProvider provider);
  object System.IConvertible.ToType(Type type, IFormatProvider provider);
  ushort System.IConvertible.ToUInt16(IFormatProvider provider);
  uint System.IConvertible.ToUInt32(IFormatProvider provider);
  ulong System.IConvertible.ToUInt64(IFormatProvider provider);
  public override string ToString();
  public string ToString(IFormatProvider provider);
  public string ToString(string format);
  public string ToString(string format, IFormatProvider provider);
+ public bool TryFormat(Span<char> destination, out Int32 charsWritten, ReadOnlySpan<char> format = default(ReadOnlySpan<char>), IFormatProvider provider = null);
+ public static bool TryParse(ReadOnlySpan<char> s, NumberStyles style, IFormatProvider provider, out Int32 result);
+ public static bool TryParse(ReadOnlySpan<char> s, out Int32 result);
  public static bool TryParse(string s, NumberStyles style, IFormatProvider provider, out Int32 result);
  public static bool TryParse(string s, out Int32 result);
}
```

SIMD

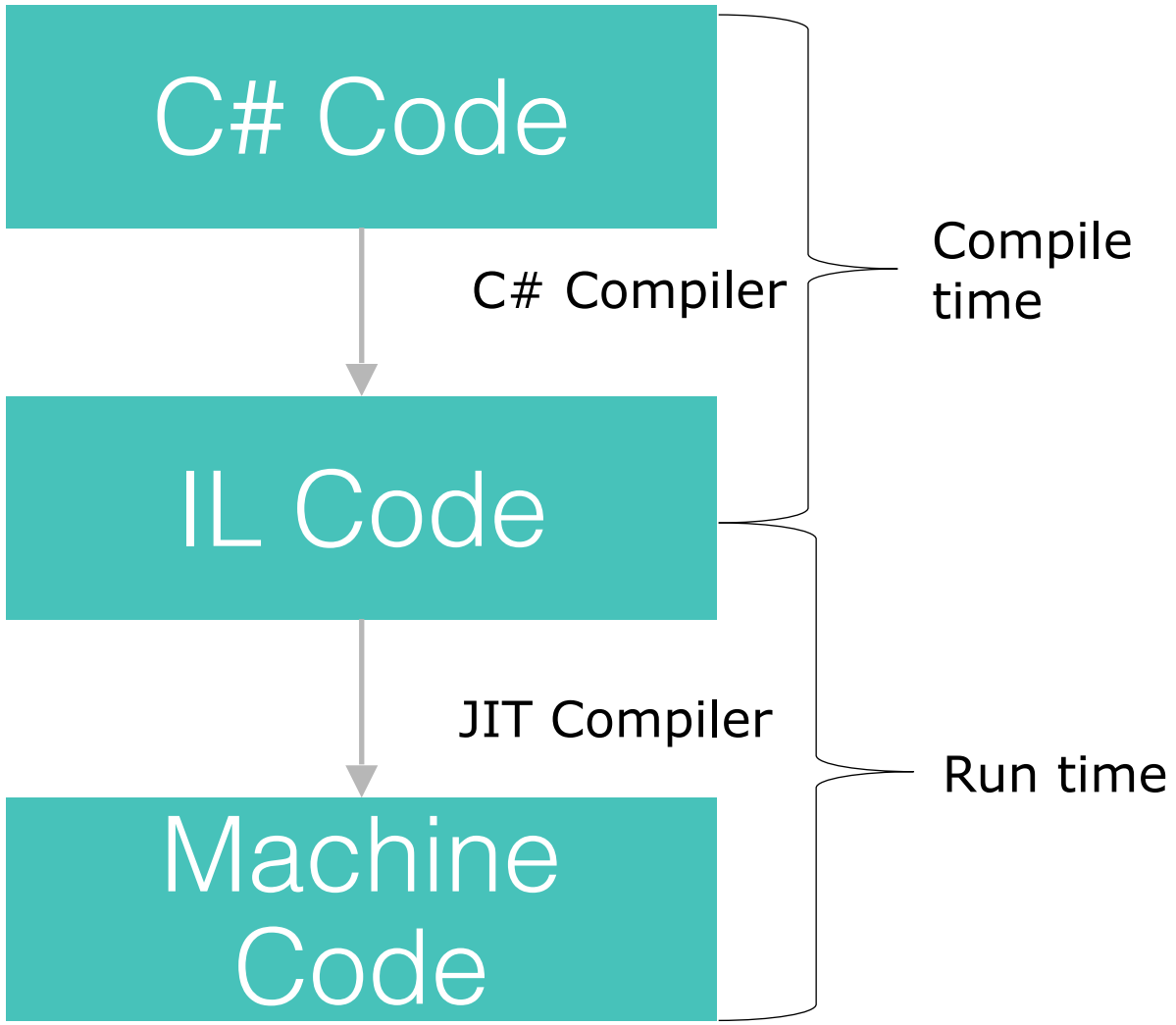
Gergely Kalapos
@gregkalapos
www.kalapos.net

SIMD

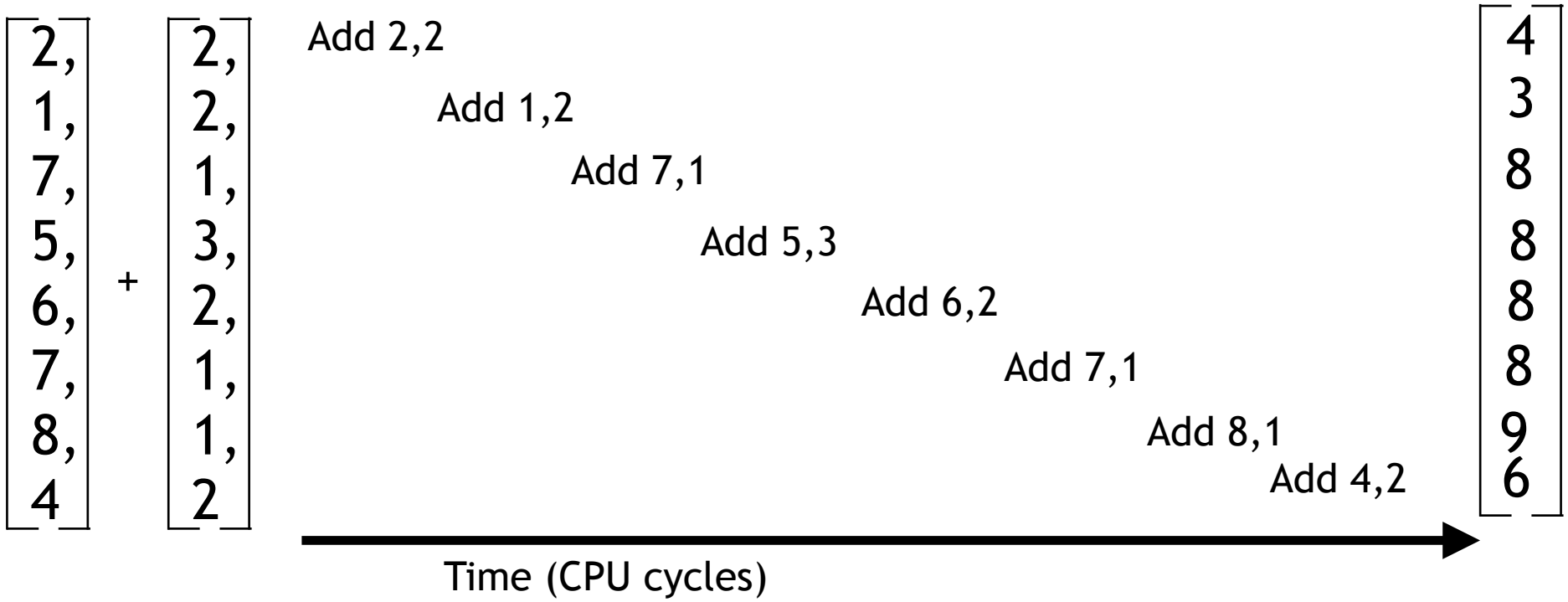
- Part of .NET Standard 2.1
- Single instruction, multiple data
- Parallelism
- Perform the same operation on multiple data points simultaneously

SIMD - Classic Assembly (no SIMD)

- add (adds 2 integers, result: 1 integer)
  ```
  add eax, ebx
  ```
- mov:
  ```
  mov ebx, count
  ```
- Takes 1 or 2 values and produces single result



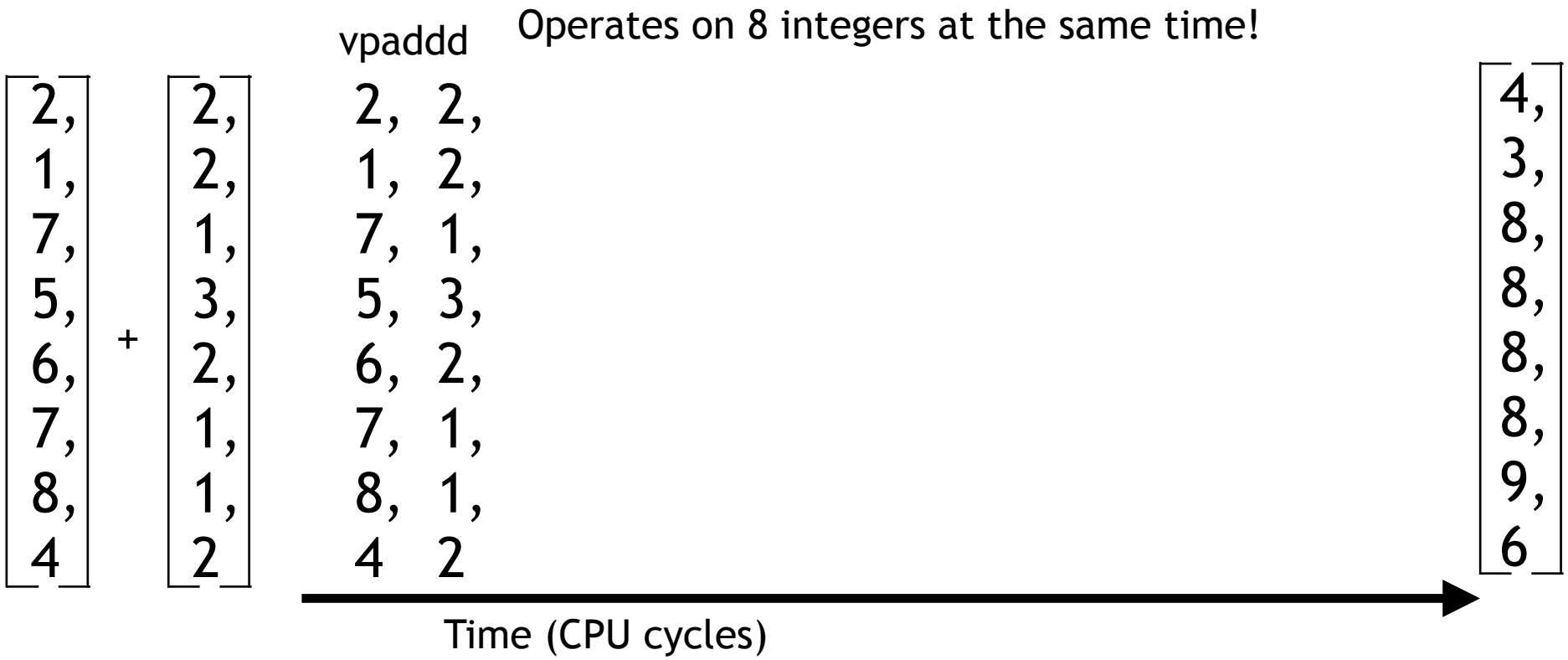
SIMD - Classic Assembly (no SIMD)



SIMD

- Single instruction, multiple data
- Those instructions can have multiple operands
- `vpaddd`: add packed doubleword integers

SIMD - add 2 arrays with vpaddd



Add 2 array, no SIMD

```
public static double[] AddArrays_Simple(double[] v1, double[] v2)
{
    double[] retVal = new double[v1.Length];

    for (int i = 0; i < v1.Length; i++)
    {
        retVal[i] = v1[i] + v2[i];
    }

    return retVal;
}
```

Add 2 array, no SIMD

```
setne   cl
movzx   ecx,cl
and     ecx,1
test    ecx,1
je      M01_L01
cmp     dword ptr [rsi+8],ebx
setge   cl
movzx   ecx,cl
cmp     dword ptr [rax+8],ebx
setge   r8b
movzx   r8d,r8b
test    r8d,ecx
je      M01_L01
                                    retVal[i] = v1[i] + v2[i];
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
IL_000d: ldloc.0
IL_000e: ldloc.1
IL_000f: ldarg.0
IL_0010: ldloc.1
IL_0011: ldelem.i4
IL_0012: ldarg.1
IL_0013: ldloc.1
IL_0014: ldelem.i4
IL_0015: add
IL_0016: stelem.i4
M01_L00
        movsxd  rcx,edx
        mov     r8d,dword ptr [rdi+rcx*4+10h]
        add     r8d,dword ptr [rsi+rcx*4+10h]
        mov     dword ptr [rax+rcx*4+10h],r8d
                                for (int i = 0; i < v1.Length; i++)
                                             ^^^
        IL_0017: ldloc.1
        IL_0018: ldc.i4.1
        IL_0019: add
        IL_001a: stloc.1
```

SIMD in .NET

- If the CPU supports SIMD, the JIT compiler emits SIMD instructions
- System.Numerics namespace

Add 2 array, with SIMD

```
public static double[] AddArrays_Vector(double[] v1, double[] v2)
{
        double[] retVal = new double[v1.Length];
        var vectSize = Vector<double>.Count;

        int i = 0;
        for (i = 0; i <= v1.Length - vectSize; i += vectSize)
        {
                var va = new Vector<double>(v1, i);
                var vb = new Vector<double>(v2, i);
                var vc = va + vb;
                vc.CopyTo(retVal, i);
        }
        if (i != v1.Length)
        {
                for (int j = i; j < v1.Length; j++)
                {
                        retVal[j] = v1[j] + v2[j];
                }
        }
        return retVal;
}
```

Add 2 array, with SIMD

```
lea     r10d,[rdx+7]
cmp     r10d,ebx
jae     00007ffc`d9351179
vmovupd ymm0,ymmword ptr [rdi+rdx*4+10h]
                                var vb = new Vector(v2, i);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
IL_001e: ldloca.s V_4
IL_0020: ldarg.1
IL_0021: ldloc.2
IL_0022: call System.Void System.Numerics.Vector`1::.ctor(!0[],System.Int32)
cmp     edx,r8d
jae     00007ffc`d9351179
cmp     r10d,r8d
vmovupd ymm1,ymmword ptr [rsi+rdx*4+10h]
                                var vc = va + vb;
    ^^^^^^^^^^^^^^^^^
IL_0027: ldloc.3
IL_0028: ldloc.s V_4
IL_002a: call System.Numerics.Vector`1 System.Numerics.Vector`1::op_Addition(System.Numerics.Vector`1,System.Numerics.Vector`1)
vpaddd  ymm0,ymm0,ymm1
                                vc.CopyTo(retVal, i);
    ^^^^^^^^^^^^^^^^^^^^
IL_0031: ldloca.s V_5
IL_0033: ldloc.0
IL_0034: ldloc.2
IL_0035: call System.Void System.Numerics.Vector`1::CopyTo(!0[],System.Int32)
cmp     edx,r9d
jae     00007ffc`d935117e
cmp     r10d,r9d
vmovupd ymmword ptr [rax+rdx*4+10h],ymm0
                        for (i = 0; i < v1.Length - vecSize; i += vecSize)
                                        ^^^^^^^^^^^^
IL_003a: ldloc.2
```

SIMD vs. Simple loop

```
BenchmarkDotNet=v0.10.13, OS=Windows 10 Redstone 3 [1709, Fall Creators Update] (10.0.16299.192)
Intel Core i5-5200U CPU 2.20GHz (Broadwell), 1 CPU, 4 logical cores and 2 physical cores
Frequency=2143475 Hz, Resolution=466.5321 ns, Timer=TSC
.NET Core SDK=2.1.4
  [Host]     : .NET Core 2.0.5 (CoreCLR 4.6.26020.03, CoreFX 4.6.26018.01), 64bit RyuJIT
  DefaultJob : .NET Core 2.0.5 (CoreCLR 4.6.26020.03, CoreFX 4.6.26018.01), 64bit RyuJIT


                     Method |      Mean |     Error |     StdDev |    Median |
--------------------------- |----------:|----------:|-----------:|----------:|
 AddArrays_Vector_Benchmark |  89.25 ns | 0.6531 ns |  0.5790 ns |  89.42 ns |
 AddArrays_Simple_Benchmark | 168.01 ns | 8.7787 ns | 25.3286 ns | 153.28 ns |

// * Hints *
Outliers
  Program.AddArrays_Vector_Benchmark: Default -> 1 outlier  was  removed
  Program.AddArrays_Simple_Benchmark: Default -> 4 outliers were removed

// * Legends *
  Mean   : Arithmetic mean of all measurements
  Error  : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  Median : Value separating the higher half of all measurements (50th percentile)
  1 ns   : 1 Nanosecond (0.000000001 sec)
```

System.IO.Pipelines

Gergely Kalapos
@gregkalapos
www.kalapos.net

System.IO.Pipelines

- A new library - System.IO.Pipelines NuGet package
- High performance IO in an easy way in .NET
- Invented by the ASP.NET Core team

Sample code: https://github.com/davidfowl/TcpEcho

Reading from a socket

```
async Task ProcessLinesAsync(NetworkStream stream)
{
        var buffer = new byte[1024];

        await stream.ReadAsync(buffer, 0, buffer.Length);

        //Process a single line
        ProcessLine(buffer);
}
```

- The entire message may not have beed received in a single call to ReadAsync
- Ignores the result of stream.ReadAsync()
- Doesn’t handle the case when multiple lines come back in a single ReadAsync call.

  Solution:
- We need to buffer the incoming data until we have found a new line.

Source: Channel9

Reading from a socket

```
async Task ProcessLinesAsync(NetworkStream stream)
{
        var buffer = new byte[1024]; var bytesBuffered = 0; var bytesConsumed = 0;

        while(true)
        {
                var bytesRead = await stream.ReadAsync(buffer, 0, buffer.Length);
                if (bytesRead == 0) { break; }

                // Keep track of the amount of buffered bytes
                bytesBuffered += bytesRead;
                var linePosition = -1;
                do
                {
                        //Look for a EOL in the buffered data
                        linePosition = Array.IndexOf(buffer, (byte)'\n', bytesConsumed, bytesBuffered - bytesConsumed);

                        if(linePosition >= 0)
                        {
                                // Calculate the length of the line based on the offset
                                var lineLength = linePosition - bytesConsumed;
                                // Process the line
                                ProcessLine(buffer, bytesConsumed, lineLength);
                                // Move the bytesConsumed to skip past the line we consumed
                                bytesConsumed += lineLength + 1;
                        }
                } while (linePosition >= 0);
        }
}
```

- Handles up to 1Kb of data
- Repeated buffer allocation

Solution:

- Resize array to fit incoming data
- Use ArrayPool<byte> to avoid repeated buffer allocation

Source: Channel9

Reading from a socket

```
async Task ProcessLinesAsync(NetworkStream stream)
{
        var buffer = ArrayPool<byte>.Shared.Rent(1024); var bytesBuffered = 0; var bytesConsumed = 0;
        while(true)
        {
                // Calculate the amount of bytes remaining in the buffer
                var bytesRemaining = buffer.Length - bytesBuffered;
                if(bytesRemaining == 0)
                {
                        // Double the buffer size and copy the previously buffered data into the new buffer
                        var newBuffer = ArrayPool<byte>.Shared.Rent(buffer.Length * 2);
                        Buffer.BlockCopy(buffer, 0, newBuffer, 0, buffer.Length);

                        // Return the old buffer to the pool
                        ArrayPool<byte>.Shared.Return(buffer);
                        buffer = newBuffer;
                        bytesRemaining = buffer.Length - bytesBuffered;
                }

                var bytesRead = await stream.ReadAsync(buffer, 0, buffer.Length);
                if (bytesRead == 0) { break; }

                // Keep track of the amount of buffered bytes
                bytesBuffered += bytesRead;
                var linePosition = -1;

                do
                {
                        //Look for a EOL in the buffered data
                        linePosition = Array.IndexOf(buffer, (byte)'\n', bytesConsumed, bytesBuffered - bytesConsumed);
                        if(linePosition >= 0)
                        {
                                // Calculate the length of the line based on the offset
                                var lineLength = linePosition - bytesConsumed;
                                // Process the line
                                ProcessLine(buffer, bytesConsumed, lineLength);
                                // Move the bytesConsumed to skip past the line we consumed
                                bytesConsumed += lineLength + 1;
                        }

                } while (linePosition >= 0);
        }
}
```

- Resizing buffers causes a lot of copying
- Allocated memory is never compacted

Stop this!!!

Source: Channel9

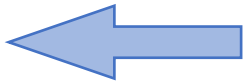
Reading from a socket - System.IO.Pipelines

```
Task ProcessLinesAsync(Socket socket)
{
        var pipe = new Pipe();
        Task writing = FillPipeAsync(socket, pipe.Writer);
        Task reading = ReadPipeAsync(pipe.Reader);

        return Task.WhenAll(reading, writing);
}
```

Source: Channel9

Reading from a socket - System.IO.Pipelines

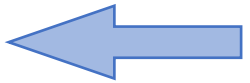
```
private async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        // Request memory from the pipe
        Memory<byte> memory = writer.GetMemory();
        try
        {
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
            {
                break;
            }
            // Tell the PipeWriter how much was read from the Socket
            writer.Advance(bytesRead);
        }
        catch (Exception e)
        {
            break;
        }

        // Make the data available to the PipeReader
        FlushResult result = await writer.FlushAsync();

        if (result.IsCompleted) { break; }
    }

    // Tell the PipeReader that there's no more data coming
    writer.Complete();
}
```



Source: Channel9

Reading from a socket - System.IO.Pipelines

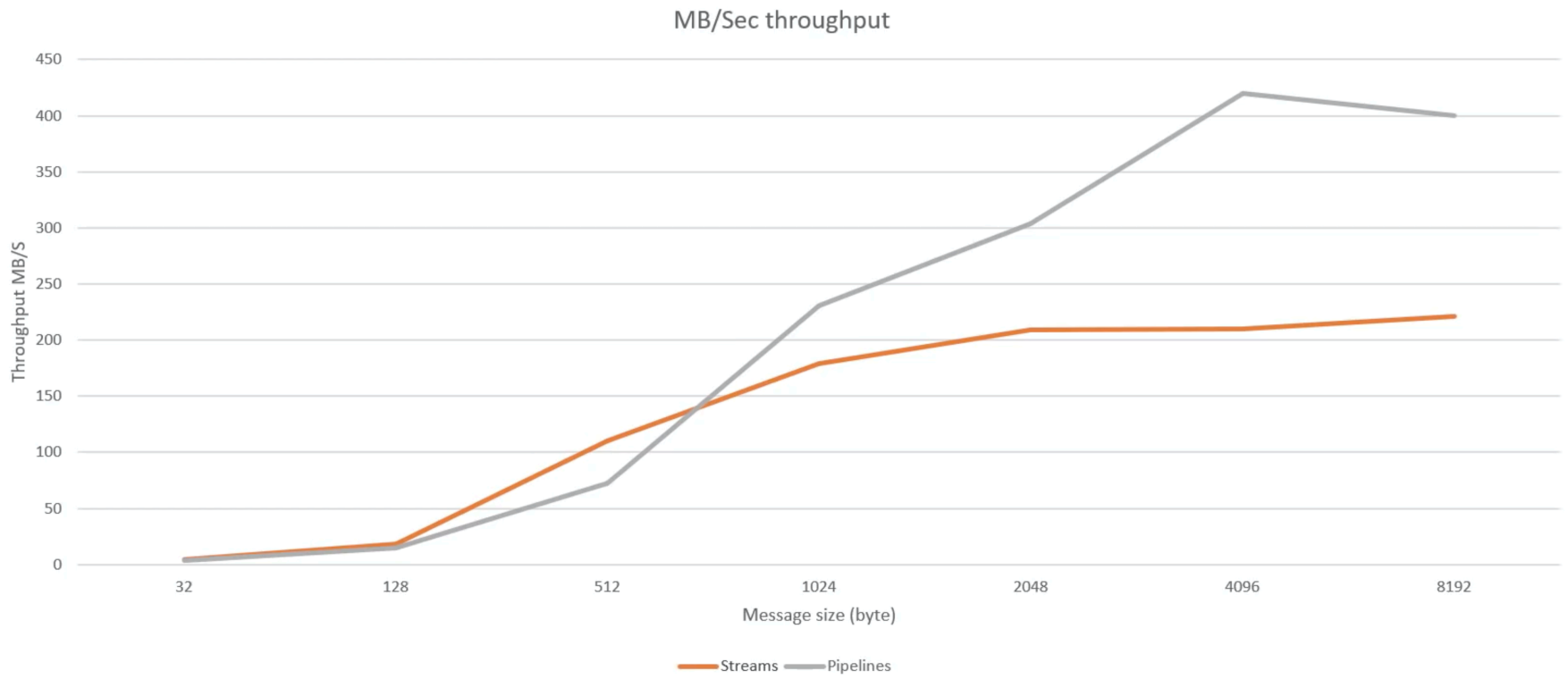
```
private async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync();

        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            // Look for a EOL in the buffer
            position = buffer.PositionOf((byte)'\n');

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip the line + the \n character
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
        } while (position != null);

        // Tell the PipeReader how much of the buffer we have consumed
        reader.AdvanceTo(buffer.Start, buffer.End);

        // Stop reading if there's no more data coming
        if (result.IsCompleted) { break; }

        // Mark the PipeReader to complete
        reader.Complete();

    }
}
```



Source: Channel9

System.IO.Pipelines - performance

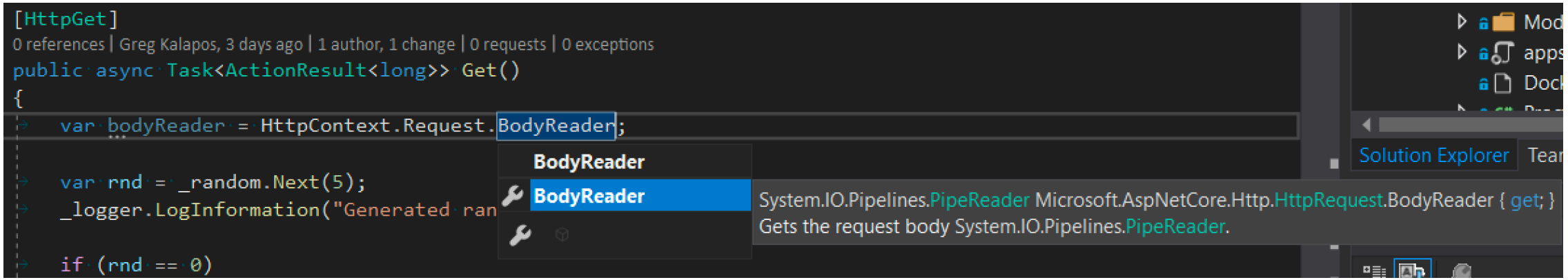
Demo app performance (naïve implementation)



Source: Channel9

System.IO.Pipelines - in the real world

- ASP.NET Core 3.0 - HttpContext.Request.BodyRequest
- SignalR



Resources

- State of the .NET Performance - Adam Sitnik (http://adamsitnik.com) https://www.youtube.com/watch?v=CSPSvBeqJ9c
- Vladimir Sadov - http://mustoverride.com
- BenchmarkDotNet: https://benchmarkdotnet.org/
- https://devblogs.microsoft.com/dotnet/system-io-pipelines-high-performance-io-in-net/
- https://channel9.msdn.com/Shows/On-NET/High-performance-IO-with-SystemIOPipelines
- Ref return and ref locals (Microsoft): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/ref-returns

The State of Observability in .NET
Tracing .NET-Application in the Cloud

- Wednesday 25. Sept. - 15:00 - 16:00
- Room: Rheingoldhalle - Forum West

Sample code + Slides

- Slides and Sample code: https://bit.ly/2ktZ2QZ
- Twitter: @gregkalapos (https://twitter.com/gregkalapos)
- Web: www.kalapos.net