State of the .NET Performance

Adam Sitnik

About myself

Work:

- Energy trading (.NET Core)
- Energy Production Optimization
- Balance Settlement
- Critical Events Detection

Open Source:

- BenchmarkDotNet (.NET Core)
- Core CLR (Spans)
- corefxlab (optimizations)
- & more

Agenda

- C# 7
  - ValueTuple
  - ref returns and locals
- .NET Core
  - Span (Slice)
  - ArrayPool
  - ValueTask
  - Pipelines (Channels)
  - Unsafe
- Supported frameworks
- Questions

ValueTuple: sample

```
(double min, double max, double avg, double sum) GetStats(double[] numbers)
{
    double min = double.MaxValue, max = double.MinValue, sum = 0;

    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] > max) max = numbers[i];
        if (numbers[i] < min) min = numbers[i];
        sum += numbers[i];
    }
    double avg = numbers.Length != 0 ? sum / numbers.Length : double.NaN;

    return (min, max, avg, sum);
}
```

ValueTuple

- Tuple which is **Value Type**:
  - less space
  - better data locality
  - NO GC
  - deterministic deallocation for stack-allocated Value Types

  You need reference to System.ValueTuple.dll

Value Types: the disadvantages?!

- Are **expensive** to copy!
- You need to study CIL and profiles to find out when it happens!

```
int result = readOnlyStructField.Method();
```

is converted to:

```
var copy = readOnlyStruct;
int result = copy.Method();
```

ref returns and locals: sample

```
ref int Max(
    ref int first, ref int second, ref int third)
{
    ref int max = ref first;

    if (first < second) max = second;
    if (second < third) max = third;

    return ref max;
}
```

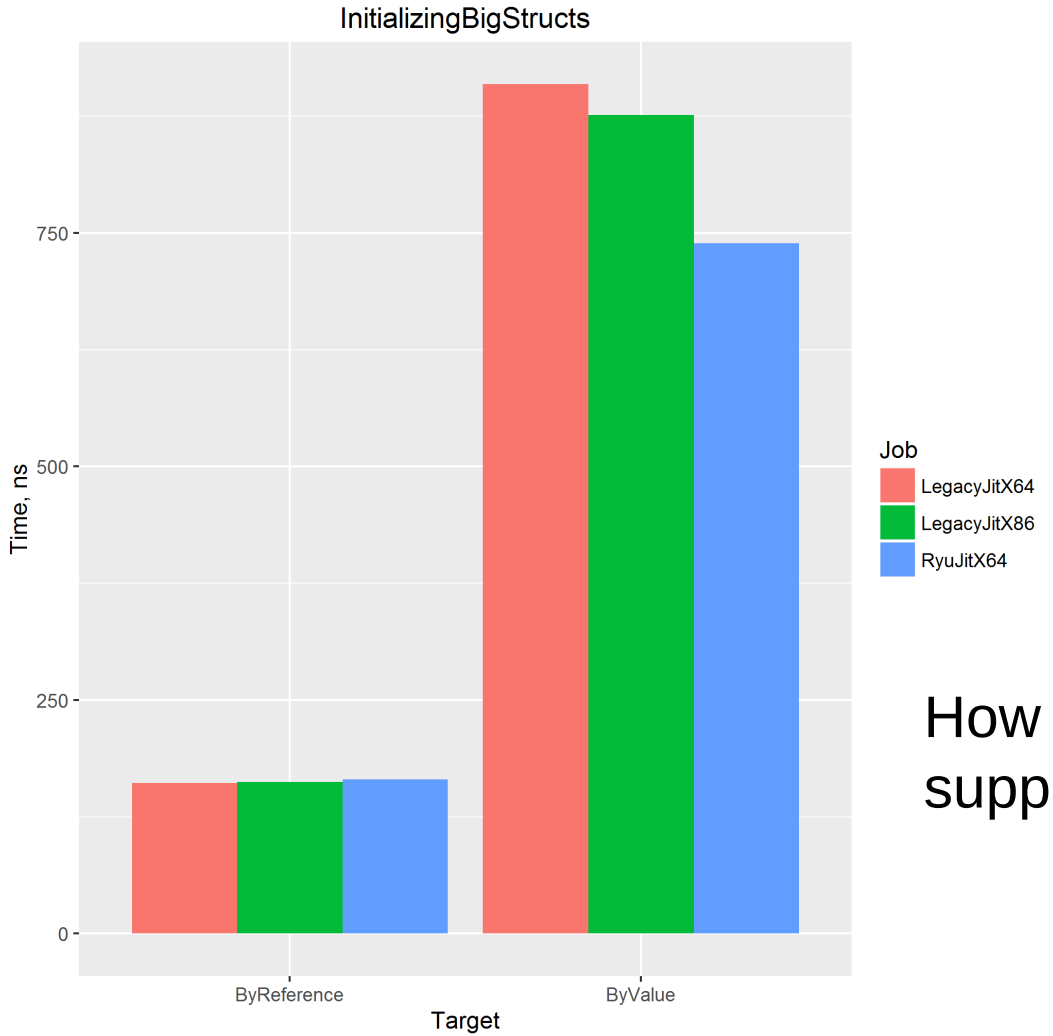
ref locals: Benchmarks: initialization

```
struct BigStruct { public int Int1, Int2, Int3, Int4, Int5; }
```

```
public void ByValue() {
    for (int i = 0; i < array.Length; i++) {
        BigStruct value = array[i];

        value.Int1 = 1;
        value.Int2 = 2;
        value.Int3 = 3;
        value.Int4 = 4;
        value.Int5 = 5;

        array[i] = value;
    }
}
```

```
public void ByReference(){
    for (int i = 0; i < array.Length; i++) {
        ref BigStruct reference = ref array[i];

        reference.Int1 = 1;
        reference.Int2 = 2;
        reference.Int3 = 3;
        reference.Int4 = 4;
        reference.Int5 = 5;

    }
}
```

Benchmark results



How can old JITs support it?

What about unsafe?!

```
void ByReferenceUnsafeExplicitExtraMethod()
{
    unsafe
    {
        fixed (BigStruct* pinned = array)
        {
            for (int i = 0;
                i < array.Length; i++)
            {
                Init(&pinned[i]);
            }
        }
    }
}
```

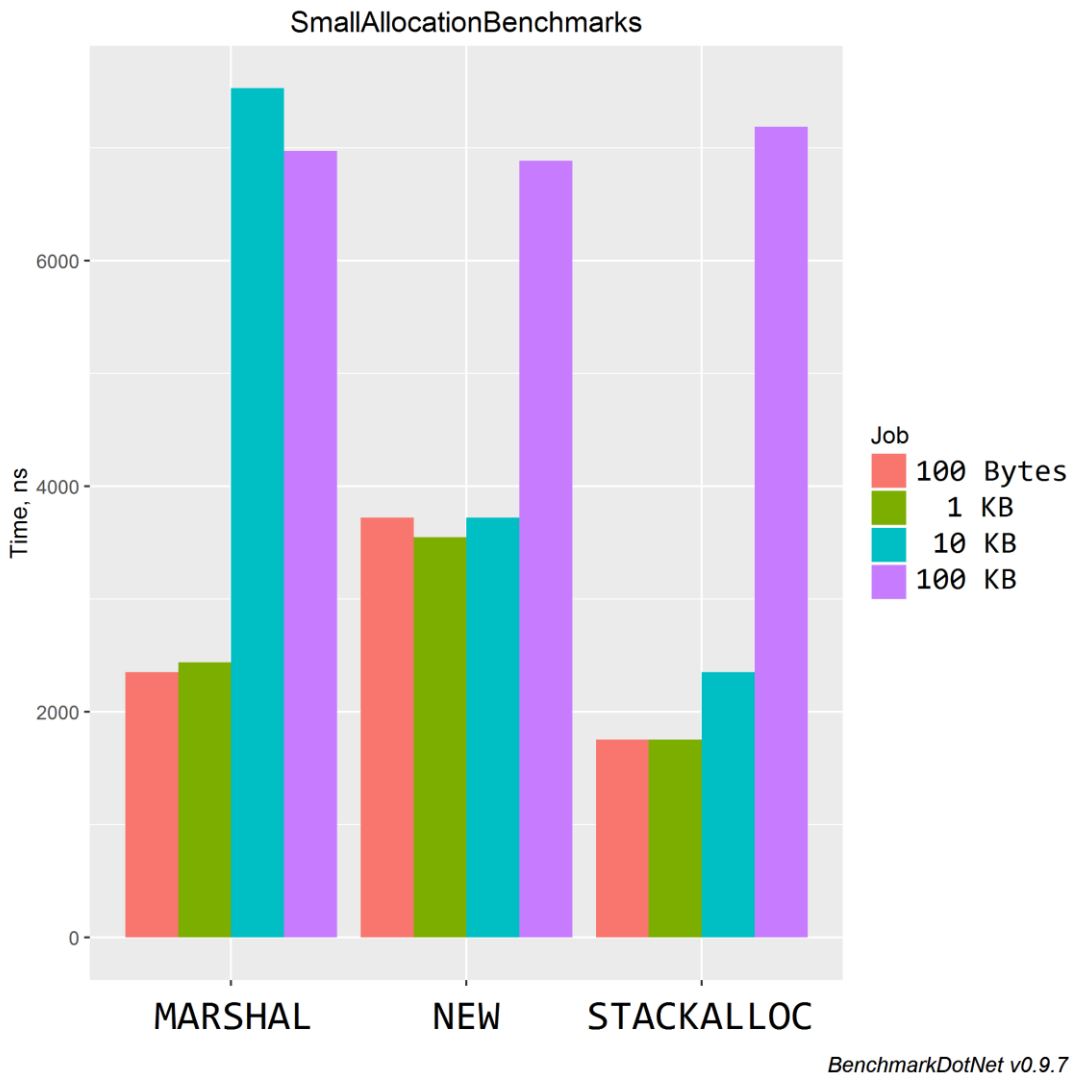
```
unsafe void Init(BigStruct* pointer)
{
    (*pointer).Int1 = 1;
    (*pointer).Int2 = 2;
    (*pointer).Int3 = 3;
    (*pointer).Int4 = 4;
    (*pointer).Int5 = 5;
}
```

Safe vs Unsafe with RyuJit

Method	Jit	Mean	Scaled
ByValue	RyuJit	742.4910 ns	4.56
ByReference	RyuJit	**162.8368 ns**	**1.00**
ByReferenceOldWay	RyuJit	170.0255 ns	1.04
ByReferenceUnsafeImplicit	RyuJit	201.4584 ns	1.24
ByReferenceUnsafeExplicit	RyuJit	200.7698 ns	1.23
ByReferenceUnsafeExplicitExtraMethod	RyuJit	171.3973 ns	1.05

Executing Unsafe code requires **full trust**. It can be a „**no go**” for **Cloud**!
No need for pinning!

Stackalloc is the fastest way to allocate small chunks of memory in .NET



	Allocation	Deallocation	Usage
Managed < 85 KB	Very cheap (NextObjPtr)	• non-deterministic • Expensive! • GC: stop the world	• Very easy • Common • Safe
Managed: LOH	Acceptable cost (free list management)	The same as above &: • Fragmentation (LOH) • LOH = Gen 2 = Full GC	
Native: Stackalloc	Very cheap	• Deterministic • Very cheap	• Unsafe • Not common • Limited
Native: Marshal	Acceptable cost (free list management)	• Deterministic • Very cheap • On demand	

Span (Slice)

It provides a uniform API for working with:

- Unmanaged memory buffers
- Arrays and subarrays
- Strings and substrings

It’s fully **type-safe** and **memory-safe**.

Almost no overhead.

It’s a Value Type.

Supports **any** memory

```
byte* pointerToStack = stackalloc byte[256];
Span<byte> stackMemory = new Span<byte>(pointerToStack, 256);
Span<byte> stackMemory = stackalloc byte[256]; // C# 8.0?

IntPtr unmanagedHandle = Marshal.AllocHGlobal(256);
Span<byte> unmanaged = new Span<byte>(unmanagedHandle.ToPointer(), 256);
Span<byte> unmanaged = Marshal.AllocHGlobal(256); // C# 8.0?

char[] array = new char[] { 'D', 'O', 'T', ' ', 'N', 'E', 'X', 'T' };
Span<char> fromArray = new Span<char>(array);
```

Single method in the API is enough

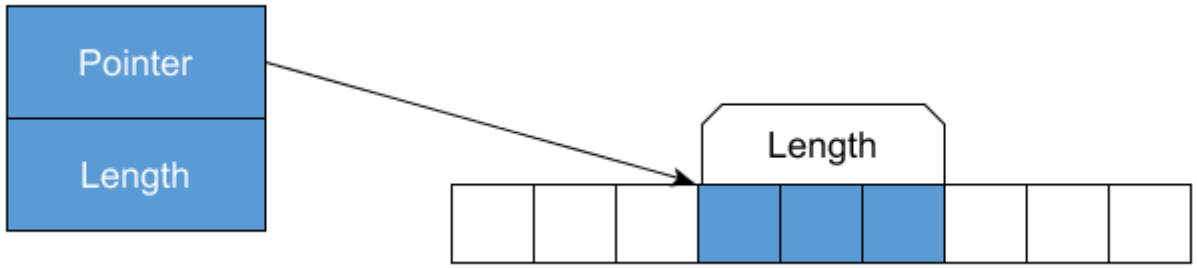
```
unsafe void Handle(byte* buffer, int length) { }

void Handle(byte[] buffer) { }


void Handle(Span<T> buffer) { }
```

Uniform access to any kind of contiguous memory

```
public void Enumeration<T>(Span<T> buffer)
{
    for (int i = 0; i < buffer.Length; i++)
    {
        Use(buffer[i]);
    }

    foreach (T item in buffer)
    {
        Use(item);
    }
}
```

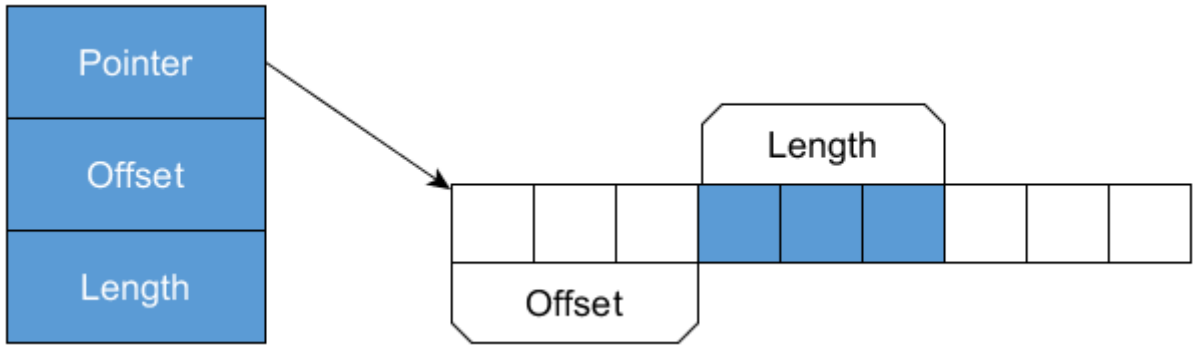
Span for new runtimes

- CoreCLR 1.2
- CLR 4.6.3? 4.6.4?



Span for existing runtimes

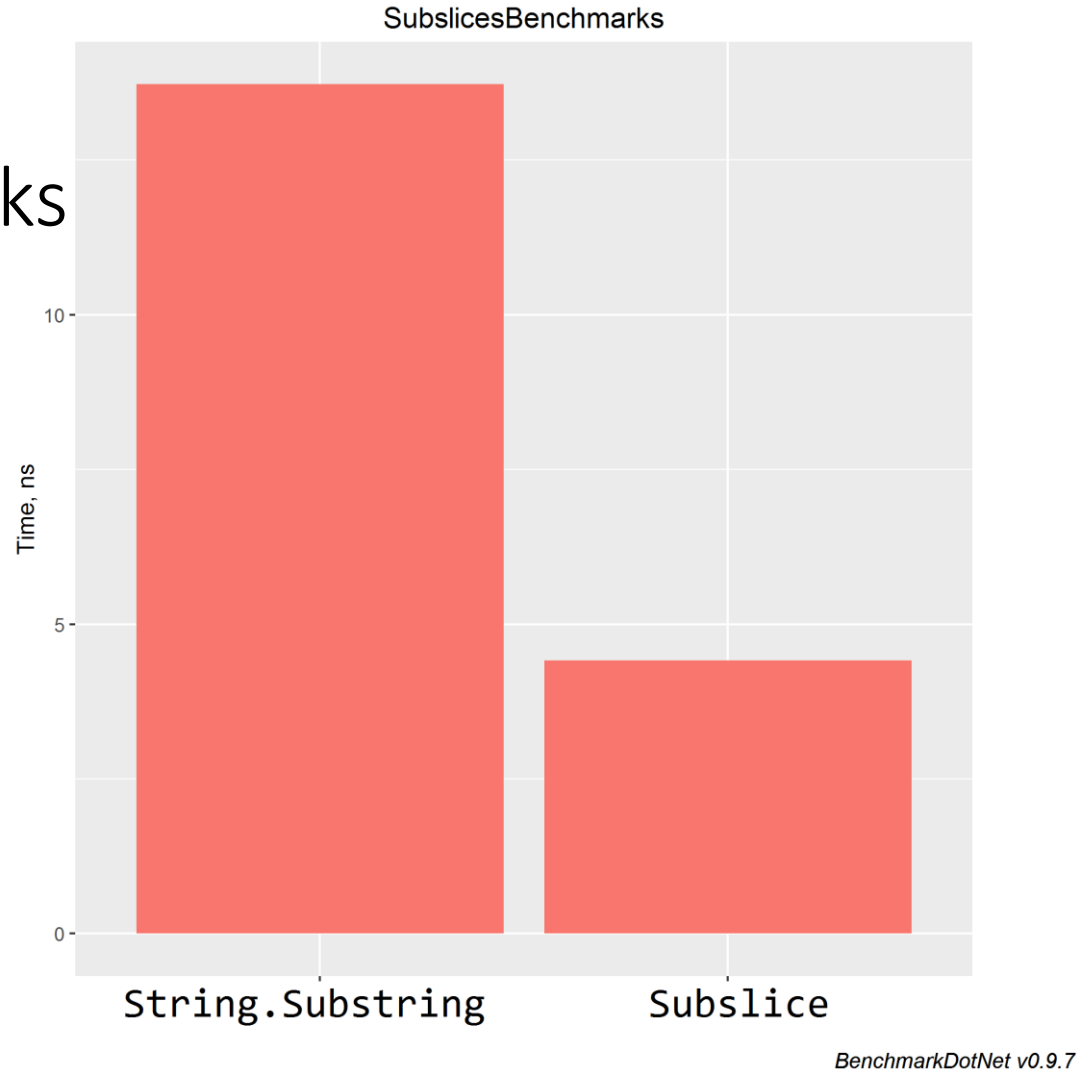
.NET 4.5+, .NET Standard 1.0



Make subslices **without** allocations

```
ReadOnlySpan<char> subslice =
    ".NET Core: Performance Storm!"
    .Slice(start: 0, length: 9);
```

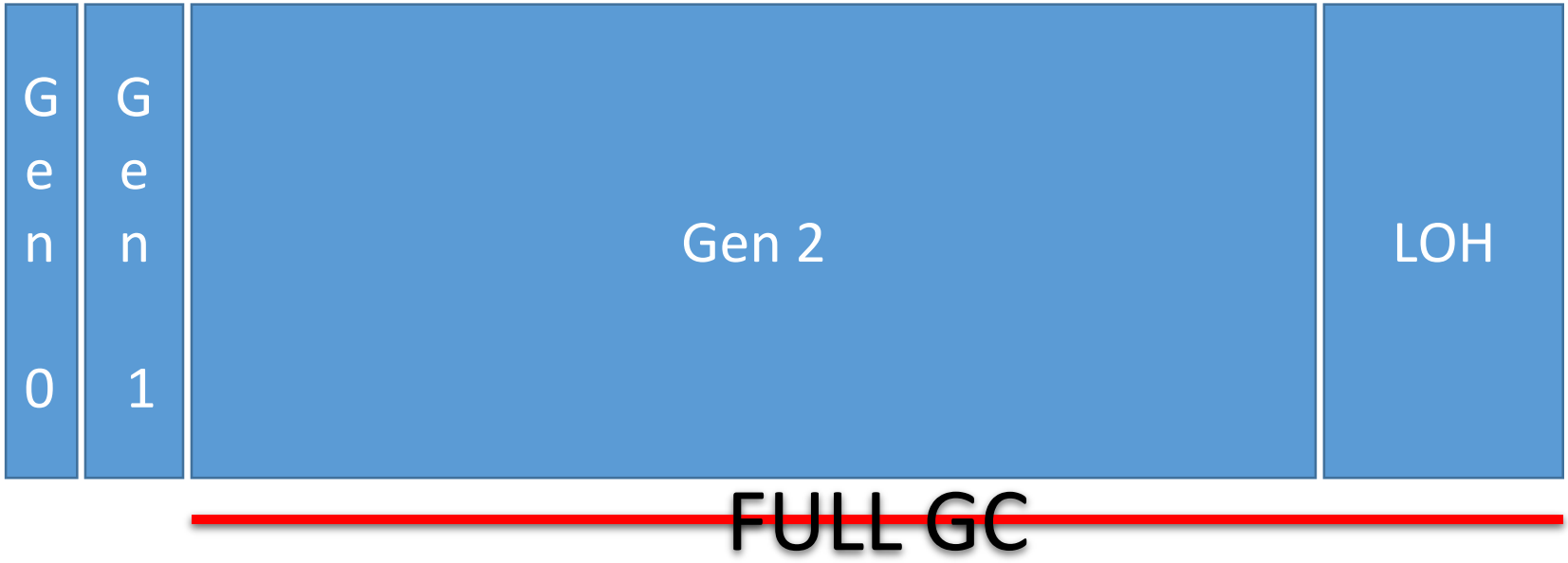
Subslice benchmarks



Possible usages

- Formatting
- Base64/Unicode encoding
- HTTP Parsing/Writing
- Compression/Decompression
- XML/JSON parsing/writing
- Binary reading/writing
- & more!!

.NET Managed Heap*



* - simplified, Workstation mode or view per logical processor in Server mode

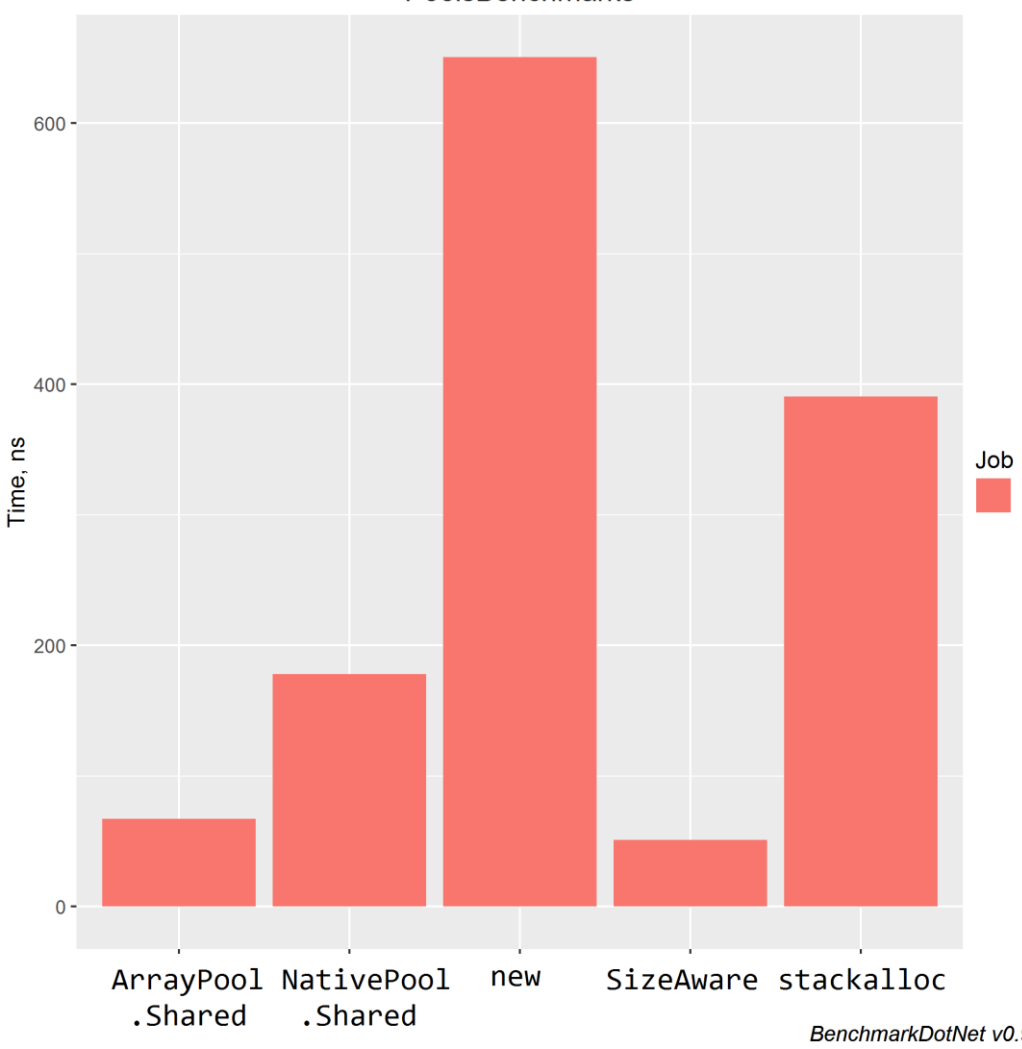
ArrayPool

- System.Buffers package
- Provides a **resource pool that enables reusing instances of T[]**
- Arrays allocated on **managed heap** with new operator
- The default maximum length of each array in the pool is 2^20 (1024*1024 = 1 048 576)

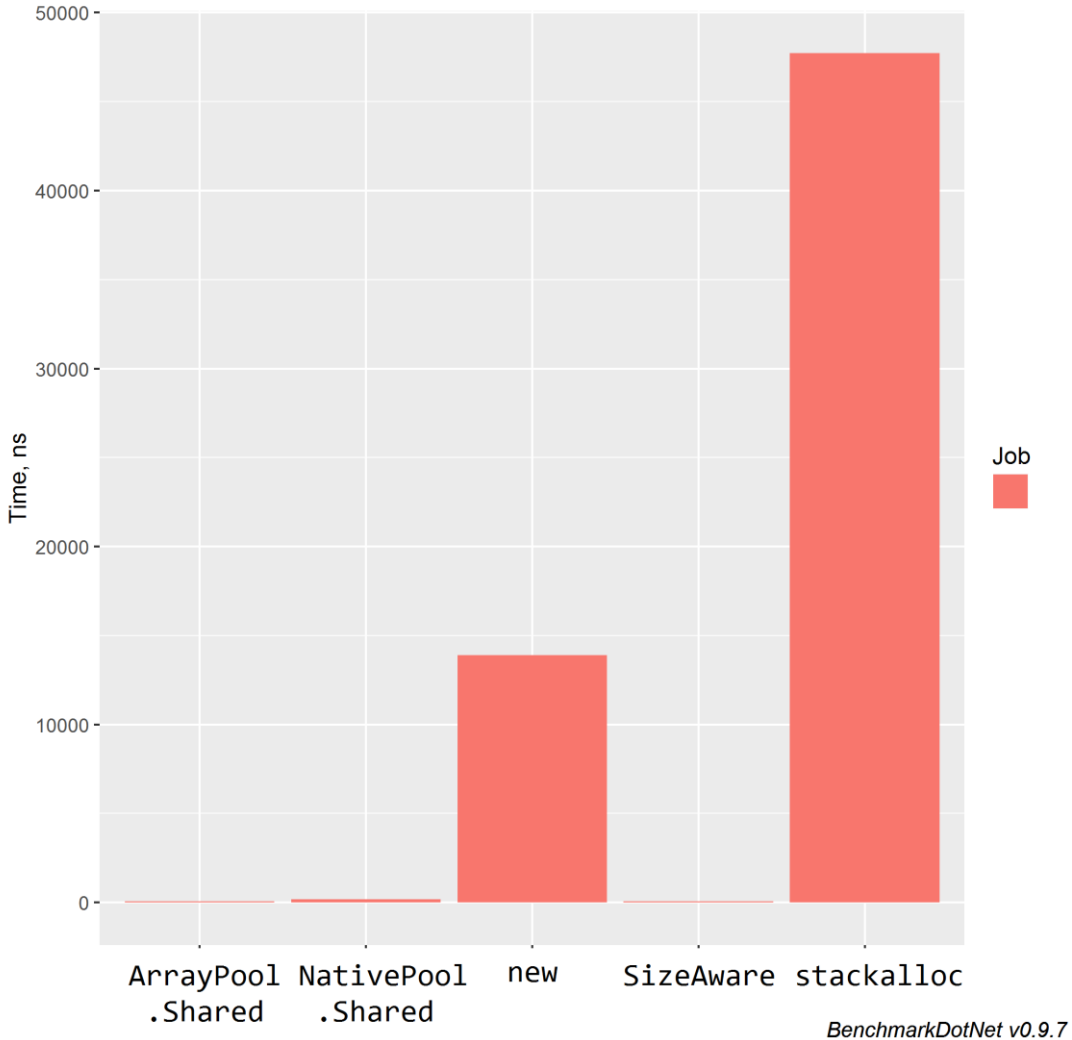
ArrayPool: Sample

```
var pool = ArrayPool<byte>.Shared;
byte[] buffer = pool.Rent(minLength);
try
{
    Use(buffer);
}
finally
{
    pool.Return(buffer);
}
```

10 KB


600
400
200
0
Time, ns
Job
ArrayPool .Shared
NativePool .Shared
new
SizeAware
stackalloc
BenchmarkDotNet v0.9.7

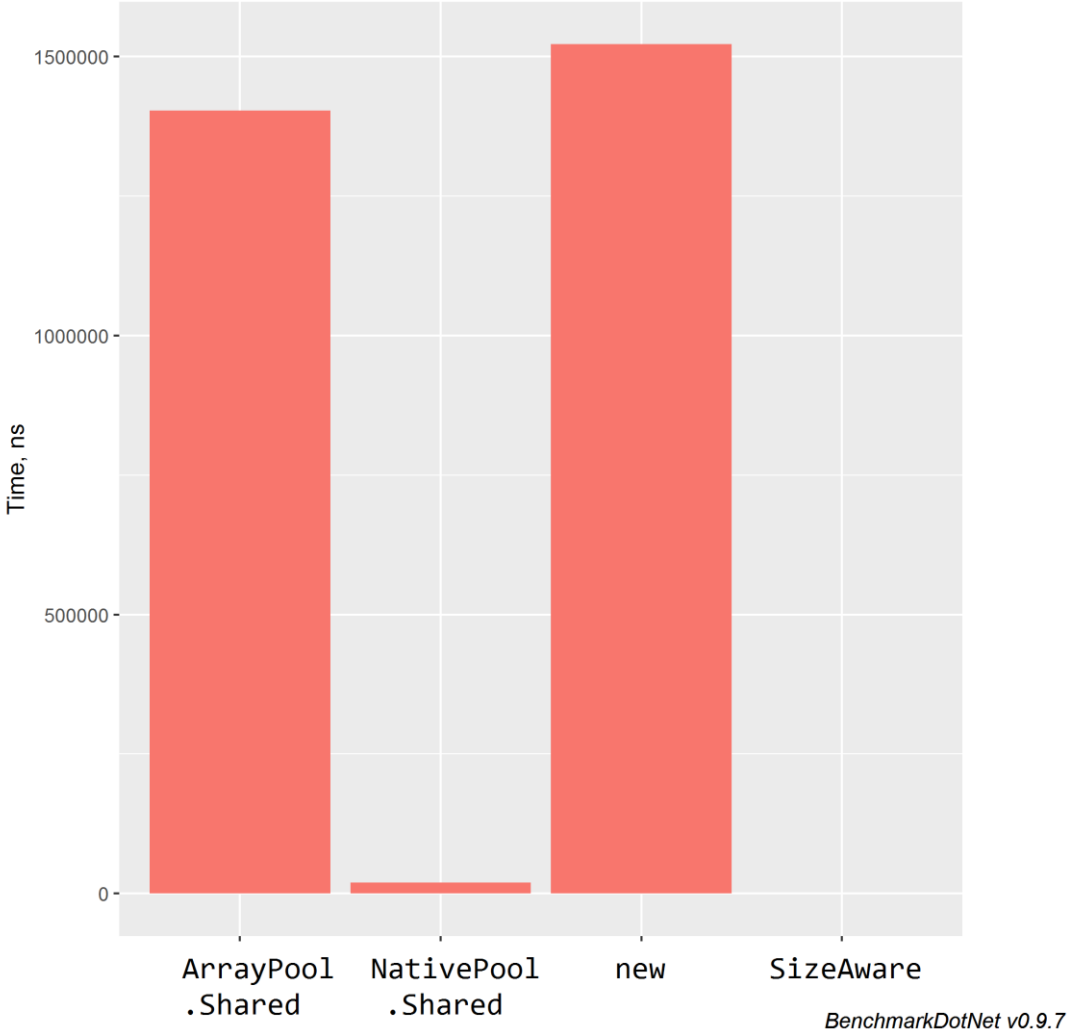
1 MB


50000
40000
30000
20000
10000
0
Time, ns
Job
ArrayPool .Shared
NativePool .Shared
new
SizeAware
stackalloc
BenchmarkDotNet v0.9.7

1 MB

Method	Median	StdDev	Scaled	Delta	Gen 0	Gen 1	Gen 2
stackalloc	51,689.8611 ns	3,343.26 ns	3.76	275.9%	-	-	-
New	13,750.9839 ns	974.0229 ns	1.00	Baseline	-	-	23 935
NativePool.Shared	186.1173 ns	12.6833 ns	0.01	-98.6%	-	-	-
ArrayPool.Shared	61.4539 ns	3.4862 ns	0.00	-99.6%	-	-	-
SizeAware	54.5332 ns	2.1022 ns	0.00	-99.6%	-	-	-

10 MB


1500000
1000000
500000
0
Time, ns
ArrayPool
.Shared
NativePool
.Shared
new
SizeAware
BenchmarkDotNet v0.9.7

Async on hotpath

```
Task<T> SmallMethodExecutedVeryVeryOften()
{
    if(CanRunSynchronously()) // true most of the time
    {
        return Task.FromResult(ExecuteSynchronous());
    }
    return ExecuteAsync();
}
```

Async on hotpath: consuming method

```
while (true)
{
     var result = await SmallMethodExecutedVeryVeryOften();
     Use(result);
}
```

ValueTask<T>: the idea

- Wraps a TResult and Task<TResult>, only **one** of which is used
- It should **not replace Task**, but help in some scenarios when:
  - method returns Task<TResult>
  - and very frequently returns **synchronously** (fast)
  - and **is invoked so often that cost of allocation of Task<TResult> is a problem**

Sample implementation of ValueTask usage

```
ValueTask<T> SampleUsage()
{
    if (IsFastSynchronousExecutionPossible())
    {
        return ExecuteSynchronous(); // INLINEABLE!!!
    }
    return new ValueTask<T>(ExecuteAsync());
}

T ExecuteSynchronous() { }

Task<T> ExecuteAsync() { }
```

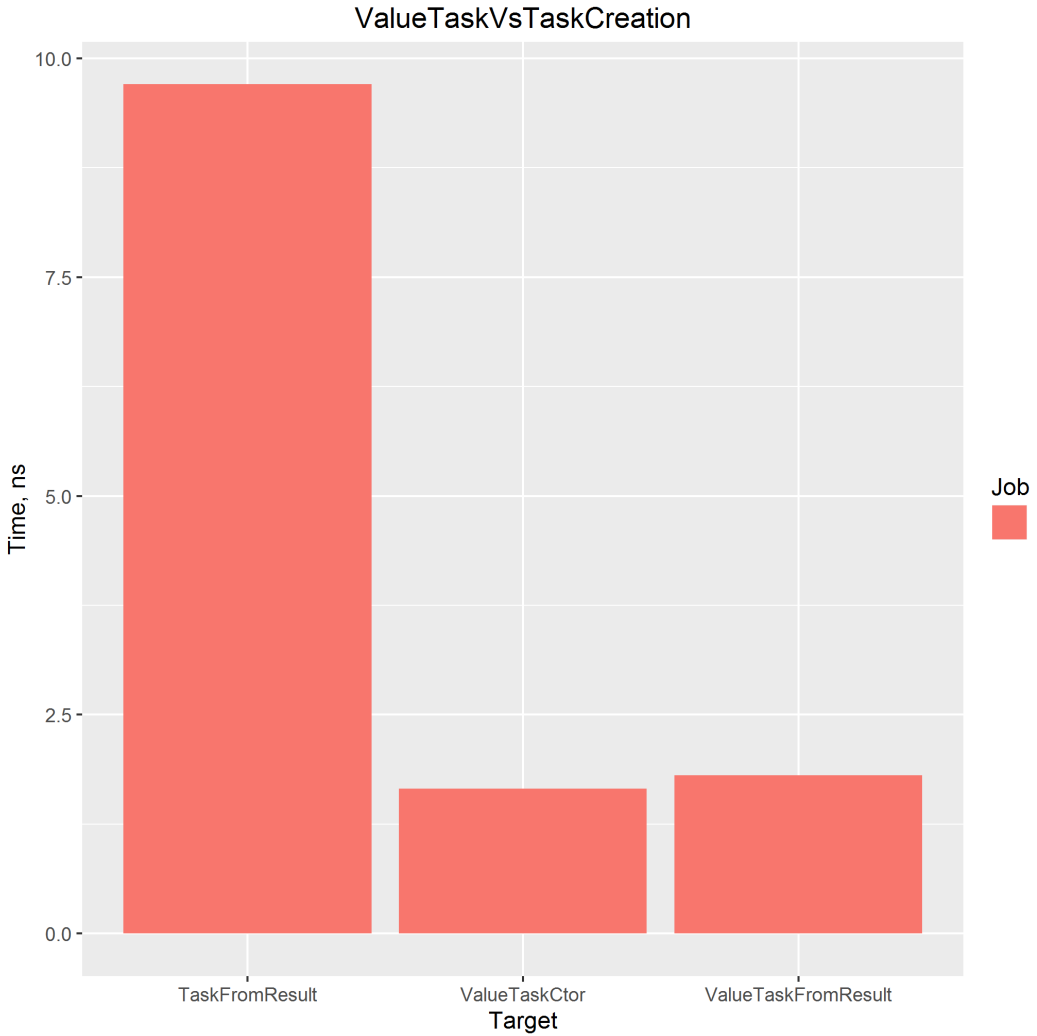
How to consume ValueTask

```
var valueTask = SampleUsage(); // INLINEABLE
if(valueTask.IsCompleted)
{
    Use(valueTask.Result);
}
else
{
    Use(await valueTask.AsTask()); // NO INLINING
}
```

ValueTask<T>: usage && gains

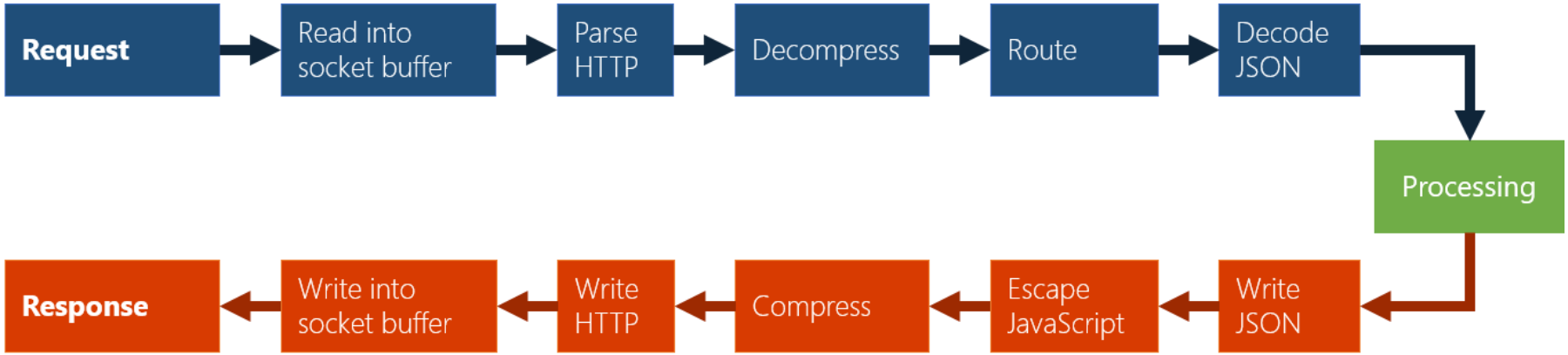
- Sample usage:
  - Sockets (already used in ASP.NET Core)
  - File Streams
  - ADO.NET Data readers
- Gains:
  - Less heap allocations
  - Method inlining is possible!
- Facts
  - Skynet 146ns for Task, 16ns for ValueTask
  - Tech Empower (Plaintext) +2.6%

ValueTask vs Task: Creation



BenchmarkDotNet v0.9.7

Web Request


Request
Read into socket buffer
Parse HTTP
Decompress
Route
Decode JSON
Processing
Write JSON
Escape JavaScript
Compress
Write HTTP
Write into socket buffer
Response

Pipelines (Channels)

- „ high performance zero-copy buffer-pool-managed asynchronous message pipes” – Marc Gravell from Stack Overflow

- Pipeline pushes data to you rather than having you pull.
- When writing to a pipeline, the caller allocates memory from the pipeline directly.
- No new memory is allocated. Only pooled memory buffer is used.

Simplified Flow

Asks for a memory buffer.

Returns pooled memory.

Starts awaiting for the data.

Writes the data to the buffer.

Reads the data from buffer.

Uses low-allocating Span based apis (parsing etc).

Returns the memory to the pool when done.

System.Runtime.CompilerServices.Unsafe

```
T As<T>(object o) where T : class;
void* AsPointer<T>(ref T value);
void Copy<T>(void* destination, ref T source);
void Copy<T>(ref T destination, void* source);
void CopyBlock(void* destination, void* source, uint byteCount);
void InitBlock(void* startAddress, byte value, uint byteCount);
T Read<T>(void* source);
int SizeOf<T>();
void Write<T>(void* destination, T value);
```

Supported frameworks

Package name	.NET Standard	.NET Framework	Release	Nuget feed
System.Slices	1.0	4.5	1.2?	Clr/fxlab
System.Buffers	1.1	4.5.1	1.0	nuget.org
System.Threading.Task.Extensions	1.0	4.5	1.0	nuget.org
System.Runtime.CompilerServices.Unsafe	1.0	4.5	1.0	corefx

Questions?

Contact:
@SitnikAdam
Adam.Sitnik@gmail.com

You can find the benchmarks at
https://github.com/adamsitnik/DotNetCorePerformance

https://github.com/adamsitnik/CSharpSevenBenchmarks