.NET Core: Performance Storm

Adam Sitnik

About myself

Work:

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

Open Source:

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

Contact:
@SitnikAdam
Adam.Sitnik@gmail.com
https://github.com/adamsitnik

Agenda

- Slice
- ArrayPool
- NativeBufferPool
- ValueTask
- Unsafe
- Supported frameworks
- Performance Radar
- Questions

Slice (Span)

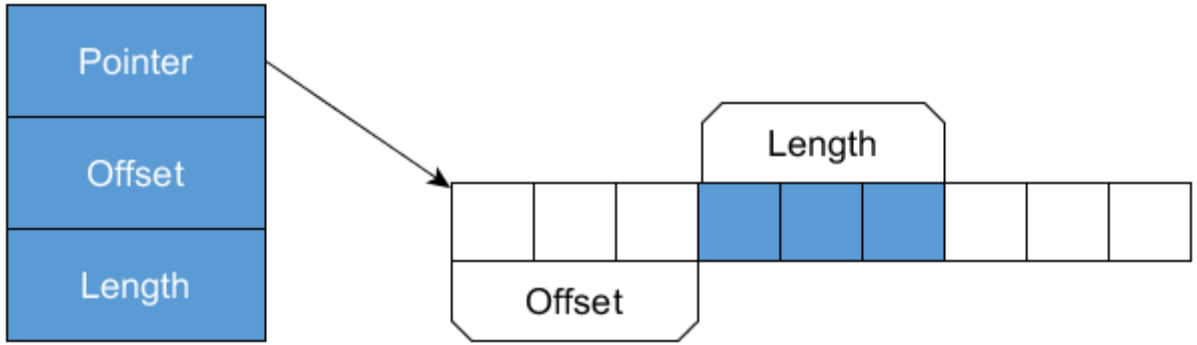
Slice is a simple struct that gives you uniform access to any kind of contiguous memory. It provides a uniform API for working with:

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

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

Almost no overhead.

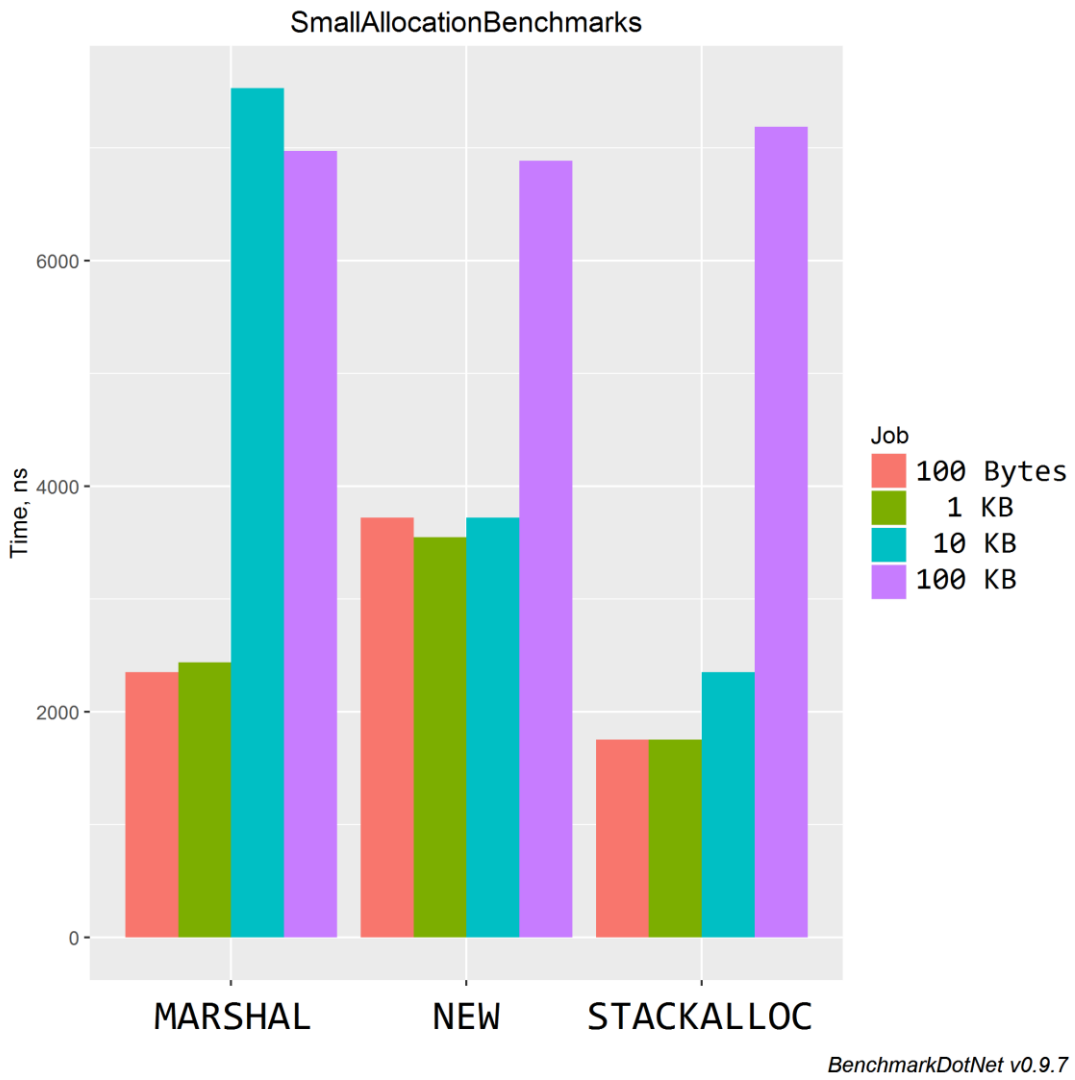
Slice: internals



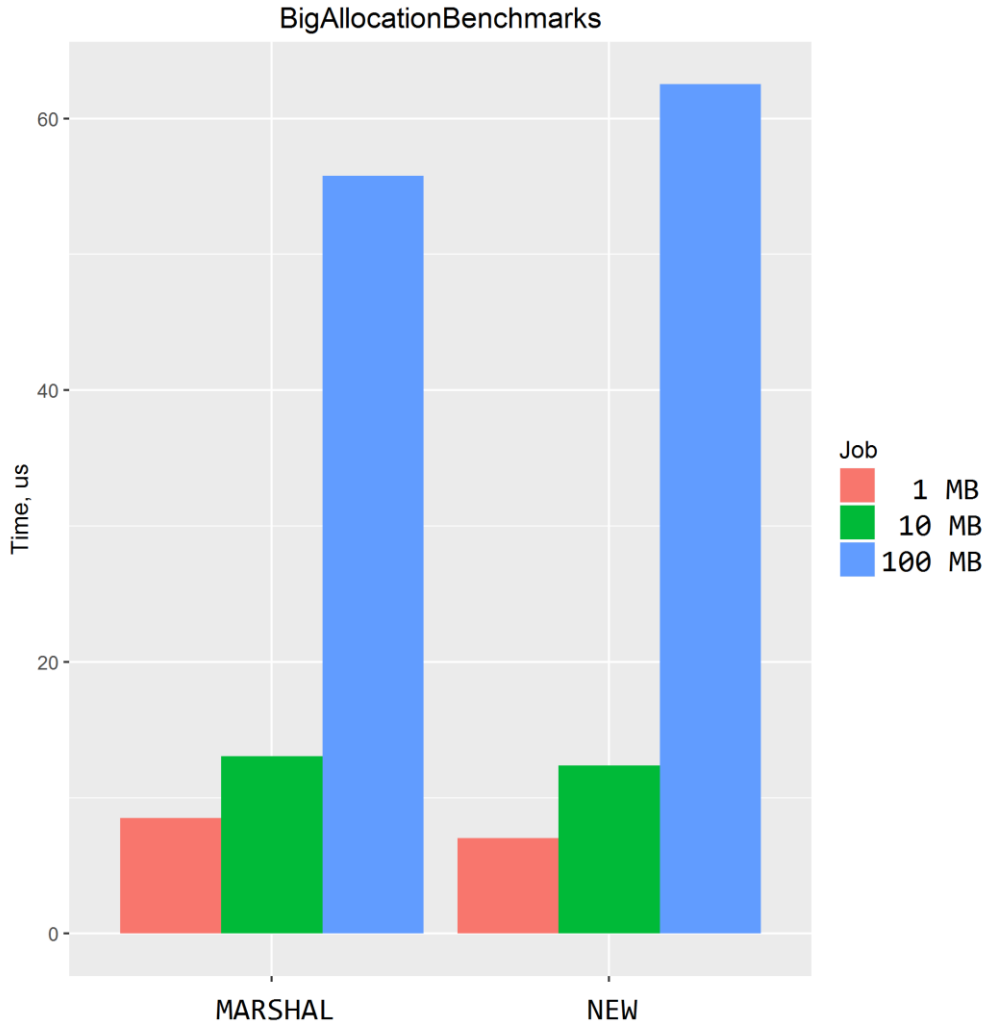
Supports unmanaged and managed memory

```
byte* pointerToStack = stackalloc byte[256];
Span<byte> stackMemory = new Span<byte>(pointerToStack, 256);

IntPtr unmanagedHandle = Marshal.AllocHGlobal(256);
Span<byte> unmanaged = new Span<byte>(unmanagedHandle.ToPointer(), 256);


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

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



Unmanaged memory => deterministic deallocation + NO GC



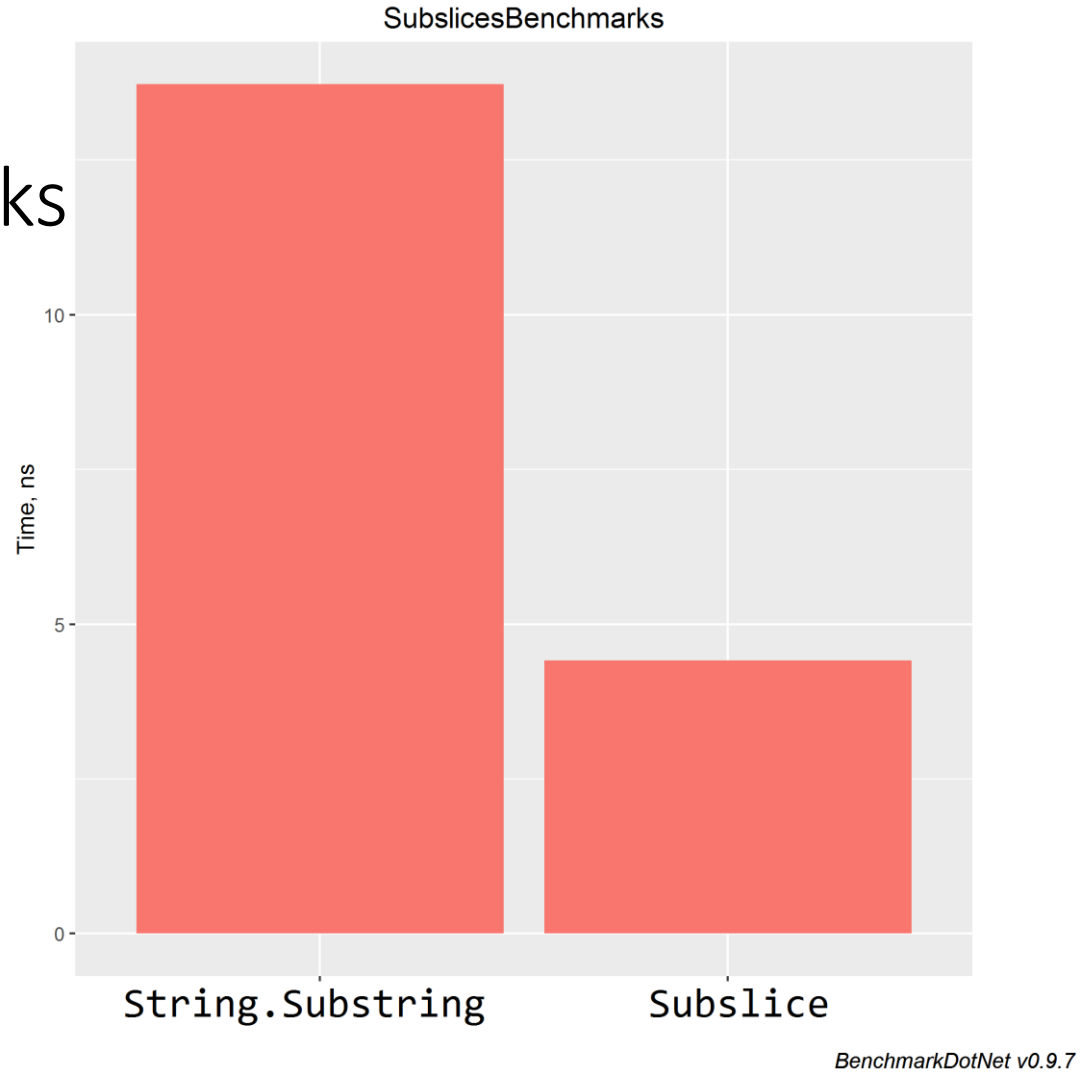
Uniform access to any kind of contiguous memory

```
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);
    }
}
```

Make subslices **without** allocations

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

Subslice benchmarks

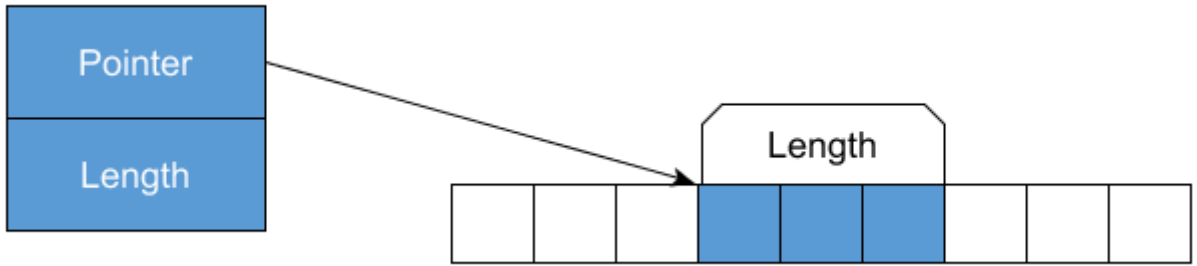


Interpret untyped data without copying or allocating

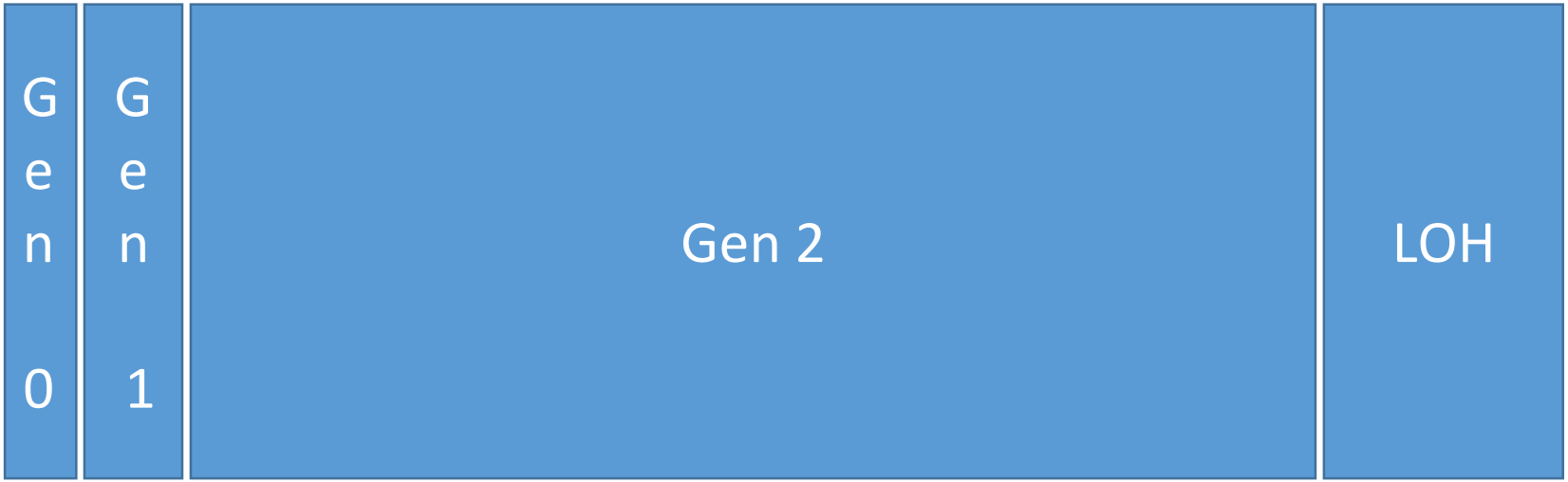
```
[StructLayout(LayoutKind.Sequential)]
struct Bid
{
    ushort Amount;
    float Price;
}

void HandleRequest(Span<byte> rawBytes)
{
    // type and memory safe that comes for free!
    Span<Bid> bids = rawBytes.Cast<byte, Bid>();
}

bool areBytewiseEqual = sliceOfOneType.BlockEquals(sliceOfOtherType);
```

Slice: Limitations

- Pointer + Offset overhead (GC compacts the heap)
- No cheap pinning (like fixed)
- For the unmanaged memory only Value Types are supported
- CIL Verification?

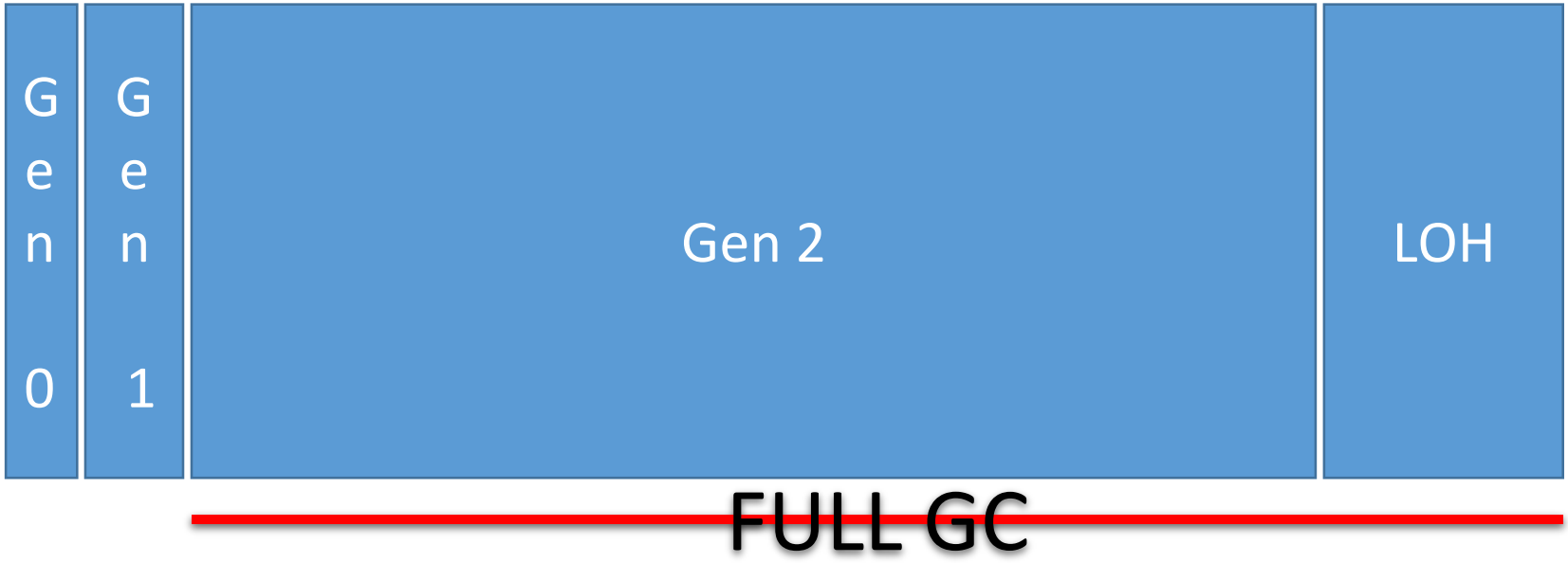


.NET Managed Heap*



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

.NET Managed Heap*



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

ArrayPool

- The idea comes from corefxlab, got to **corefx**
- System.Buffers package, part of **RC2**
- 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)

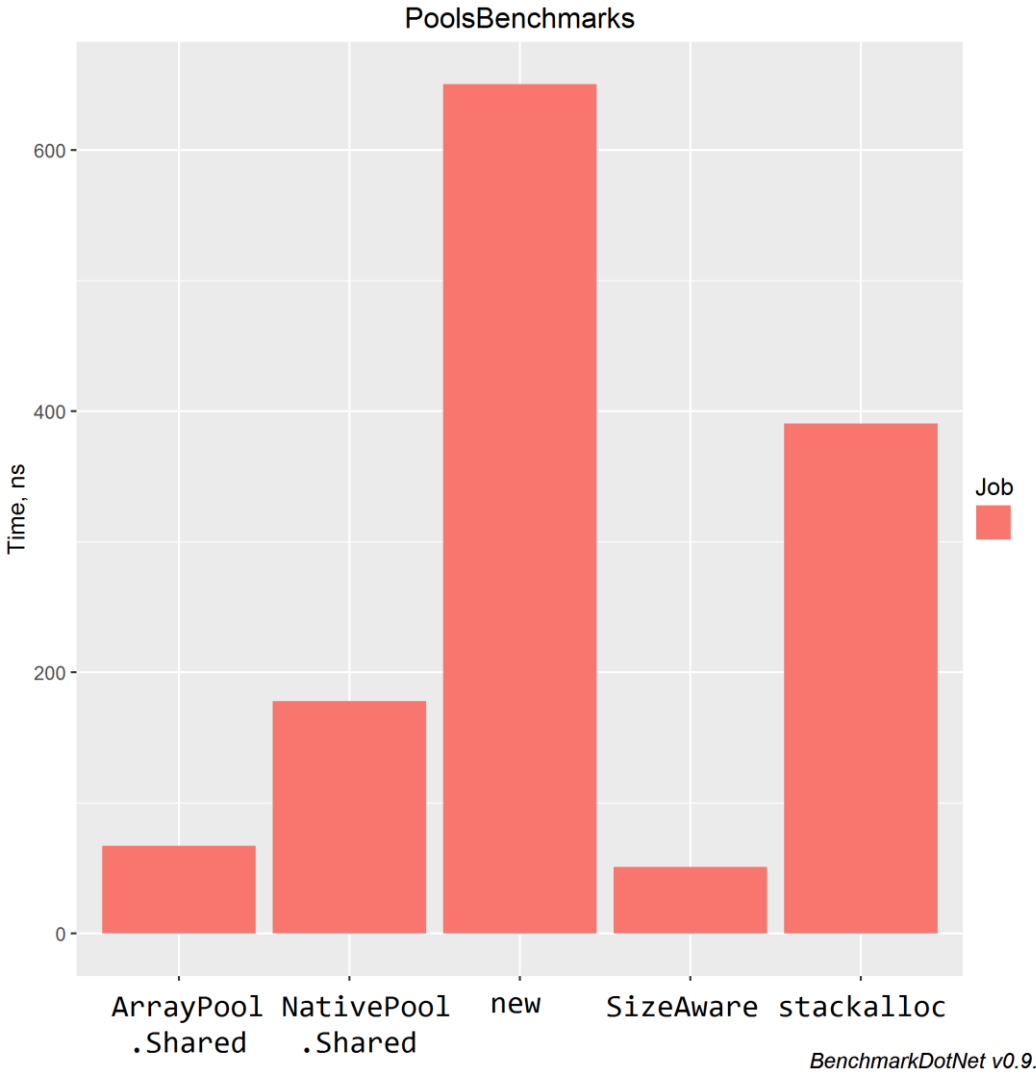
NativeBufferPool

- The idea comes from corefxlab
- Most probably will move **together with Slices** to corefx
- System.Buffers.Experimental package, corefxlab feed only
- Pool of **Slices<T>**
- **Unmanaged memory**, allocated with Marshal.AllocHGlobal
- The **default for max size is 2MB,** taken as the default since the average HTTP page is 1.9MB

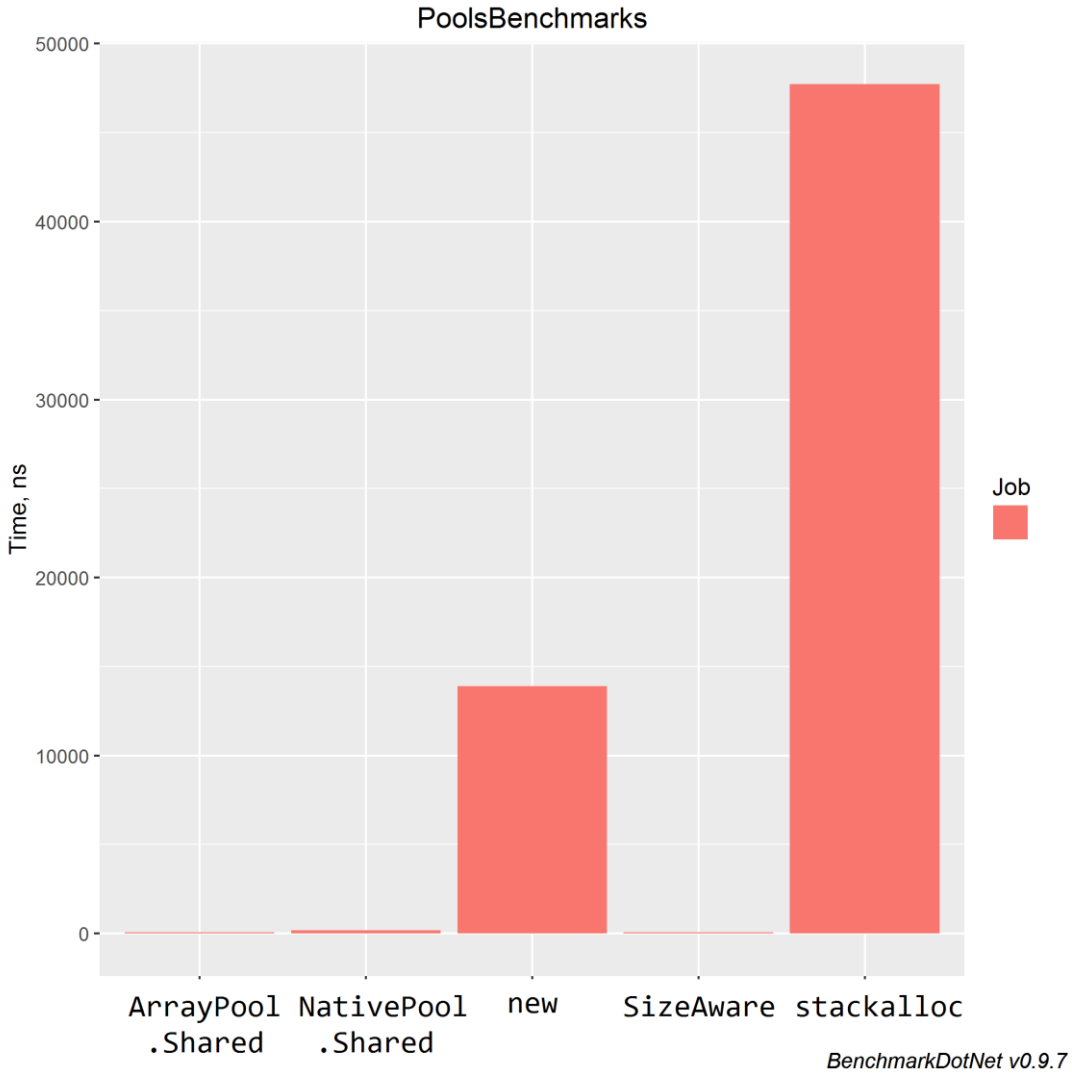
How to work with Pools

- Get your own instance
  - `ArrayPool<T>.Shared` (**Thread safe**)
  - `NativeBufferPool<byte>.SharedByteBufferPool` (**TS**)
  - `ArrayPool<T>.Create(int maxArrayLength, ..);`
  - Derive custom class from ArrayPool
- Rent
  - Specify the min size
  - Bigger can be returned, don’t rely on .Length
  - If min size > maxArrayLength **new instance will be returned**
- Finally { Return }

10 KB


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

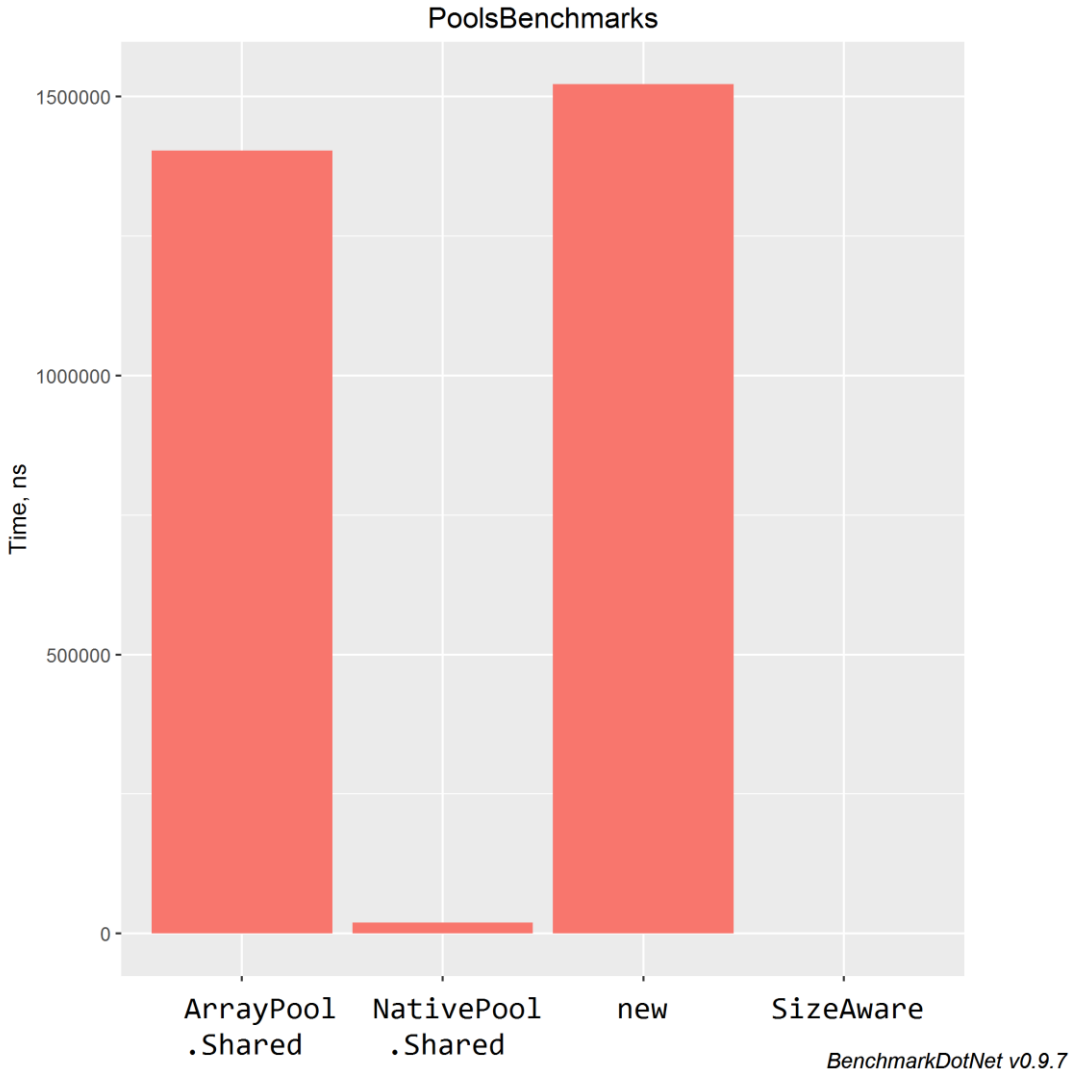
1 MB


PoolsBenchmarks
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


PoolsBenchmarks
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();
}
```

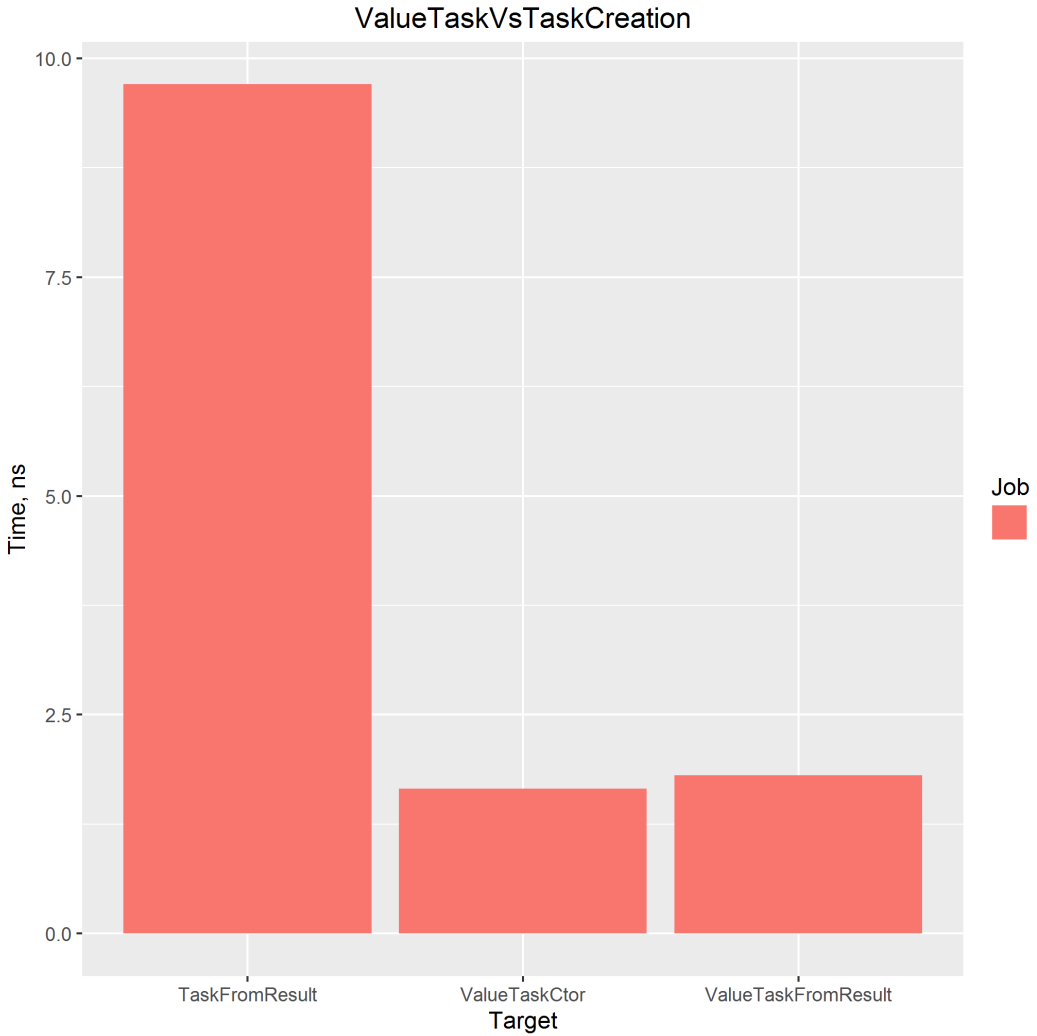
Async on hotpath: consuming method

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

ValueTask<T>

- The idea comes from **corefxlab**, got to **corefx**
- System.Threading.Tasks.Extensions package, part of RC2
- **Is Value type:**
  - **less space**
  - **less heap allocations = NO GC**
  - **probably faster memory reads**
  - **better data locality**

ValueTask vs Task: Creation



BenchmarkDotNet v0.9.7

ValueTask vs Task: Creation

Method	Median	StdDev	Scaled	Delta	Gen 0	Gen 1	Gen 2
Task.FromResult	9.6268 ns	0.1796 ns	1.00	Baseline	9 793	-	-
new ValueTask	1.6605 ns	0.0567 ns	0.17	-82.8%	-	-	-
ValueTask.From Result	1.7420 ns	0.1672 ns	0.18	-81.9%	-	-	-

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%

System.Runtime.CompilerServices.Unsafe

- Provides the System.Runtime.CompilerServices.Unsafe class, which provides generic, low-level functionality for manipulating pointers.
- Very similar to https://github.com/DotNetCross/Memory.Unsafe
- Bleeding edge, +- 4 weeks old ;)
- Not in RC2, will be part of RTM, currently only corefx feed
- Whole code is single IL file

Unsafe class api

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

Supported frameworks

Package name	.NET Standard	.NET Framework	Release	Nuget feed
System.Slices	1.0	4.5	RTM?	corefxlab
System.Buffers	1.1	4.5	RC2	nuget.org
System.Buffers.Experimental	1.1	4.5	RTM?	corefxlab
System.Threading.Task.Extensions	1.0	4.5	RC2	nuget.org
System.Runtime.CompilerServices.Unsafe	1.0	4.5	RTM	corefx

.NET Core Performance Radar

Adopt

- ArrayPool
- Zlib

Trial

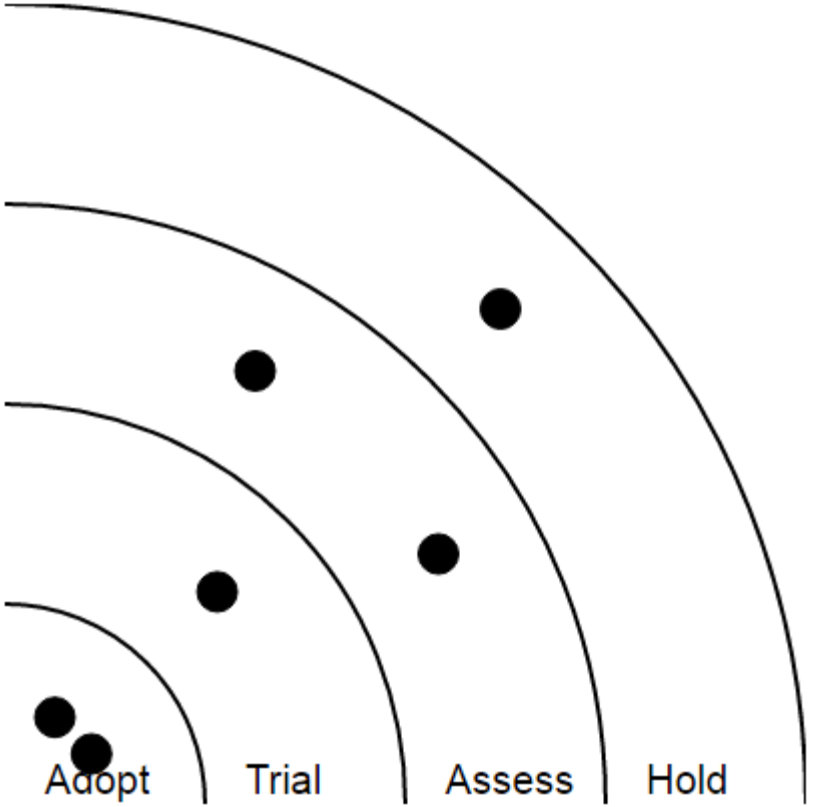
- ValueTask

Assess

- NativePool
- Slice

Hold

- Utf8String



Questions?

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