.NET Core: Performance Revolution

Adam Sitnik

About myself

Open Source:

- BenchmarkDotNet
- Awesome .NET Performance
- Core CLR (Span<T>)
- CoreFx Lab (Span<T>)
- & more

Work:

- Energy Trading

(.NET Core running in production since July 2016)

ASP.NET Core road to high-performance

- Performance Improvements in .NET Core
- Performance Improvements in RyuJIT
- .NET Native & CoreRT (later today)
- New tools:
  - Span<T>
  - ArrayPool<T>
  - ValueTask<T>
  - Unsafe

How to avoid GC?

- Reduce allocations
- Eliminate all managed allocations:
  - Pool the memory
  - Use unmanaged memory

Async on hotpath

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

Sample ValueTask usage

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
ValueTask<int> SampleUsage()
    => IsFastSynchronousExecutionPossible()
        ? new ValueTask<int>(
            result: ExecuteSynchronous()) // INLINEABLE!!!
        : new ValueTask<int>(
            task: ExecuteAsync());

int ExecuteSynchronous() { }
Task<int> ExecuteAsync() { }
```

How **not** to consume ValueTask

```
async ValueTask<int> ConsumeWrong(int repeats)
{
    int total = 0;
    while (repeats-- > 0)
        total += await SampleUsage();

    return total;
}
```

Async Task Method Builder

```
[AsyncStateMachine(typeof(DemoInt.<ConsumeWrong>d__4))]
private Task ConsumeWrong(int repeats)
{
    DemoInt.<ConsumeWrong>d__4 <ConsumeWrong>d__;
    <ConsumeWrong>d__.<>4__this = this;
    <ConsumeWrong>d__.repeats = repeats;
    <ConsumeWrong>d__.<>t__builder = AsyncTaskMethodBuilder.Create();
    <ConsumeWrong>d__.<>1__state = -1;
    AsyncTaskMethodBuilder <>t__builder = <ConsumeWrong>d__.<>t__builder;
    <>t__builder.Start<DemoInt.<ConsumeWrong>d__4>(ref <ConsumeWrong>d__);
    return <ConsumeWrong>d__.<>t__builder.Task;
}
```

How to consume ValueTask

```
async ValueTask<int> ConsumeProperly(int repeats)
{
    int total = 0;
    while (repeats-- > 0)
    {
        ValueTask<int> valueTask = SampleUsage(); // INLINEABLE

        total += valueTask.IsCompleted
            ? valueTask.Result // hot path
            : await valueTask.AsTask();
    }

    return total;
}
```

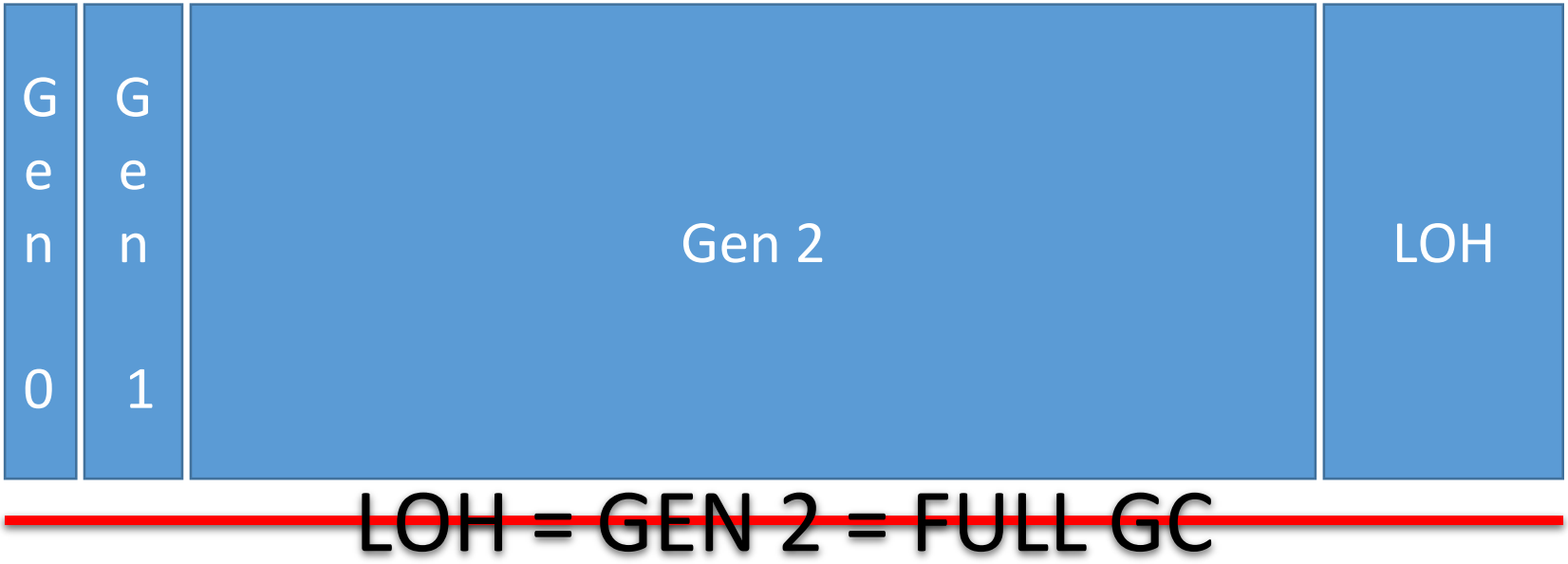
ValueTask vs Task: Overhead Only

Method	Repeats	Mean	Scaled	Gen 0	Gen 1	Allocated
Task	100	720.9 ns	1.49	3.4674	0.0001	7272 B
ValueTask_Wrong	100	1,097.4 ns	2.27	-	-	0 B
ValueTask_Properly	100	482.9 ns	1.00	-	-	0 B

Value Task: Summary

- It’s not about replacing Task
- It has a **single purpose**: reduce heap allocations in async hot path where common synchronous execution is possible
- You can benefit from inlining, but not for free
- Use the .IsCompleted and .Result for getting best performance

.NET Managed Heap*



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

ArrayPool

- **Pool of reusable managed arrays**
- The default maximum length of each array in the pool is 2^20 (1024*1024 = 1 048 576)
- System.Buffers package

ArrayPool: Sample

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

Allocate? Pool?

Method	Size	Mean	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Allocate	100	8.149 ns	0.0169 ns	0.0610	-	-	128 B
RentAndReturn_Shared	100	**43.446 ns**	0.0908 ns	-	-	-	**0 B**

Allocate? Pool?

Method	Size	Mean	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Allocate	100	8.149 ns	0.0169 ns	0.0610	-	-	128 B
RentAndReturn_Shared	100	**43.446 ns**	0.0908 ns	-	-	-	**0 B**
Allocate	1 000	41.122 ns	0.0812 ns	0.4880	0.0000	-	1024 B
RentAndReturn_Shared	1 000	**42.535 ns**	0.0621 ns	-	-	-	**0 B**

Allocate? Pool?

Method	Size	Mean	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Allocate	100	8.149 ns	0.0169 ns	0.0610	-	-	128 B
RentAndReturn_Shared	100	**43.446 ns**	0.0908 ns	-	-	-	**0 B**
Allocate	1 000	41.122 ns	0.0812 ns	0.4880	0.0000	-	1024 B
RentAndReturn_Shared	1 000	**42.535 ns**	0.0621 ns	-	-	-	**0 B**
Allocate	10 000	371.113 ns	3.2994 ns	4.7847	0.0000	-	10024 B
RentAndReturn_Shared	10 000	**42.565 ns**	0.0450 ns	-	-	-	**0 B**

Allocate? Pool?

Method	Size	Mean	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Allocate	100	8.149 ns	0.0169 ns	0.0610	-	-	128 B
RentAndReturn_Shared	100	**43.446 ns**	0.0908 ns	-	-	-	**0 B**
Allocate	1 000	41.122 ns	0.0812 ns	0.4880	0.0000	-	1024 B
RentAndReturn_Shared	1 000	**42.535 ns**	0.0621 ns	-	-	-	**0 B**
Allocate	10 000	371.113 ns	3.2994 ns	4.7847	0.0000	-	10024 B
RentAndReturn_Shared	10 000	**42.565 ns**	0.0450 ns	-	-	-	**0 B**
Allocate	100 000	3,625.029 ns	17.2533 ns	31.2497	31.2497	31.2497	100024 B
RentAndReturn_Shared	100 000	**42.426 ns**	0.0555 ns	-	-	-	**0 B**

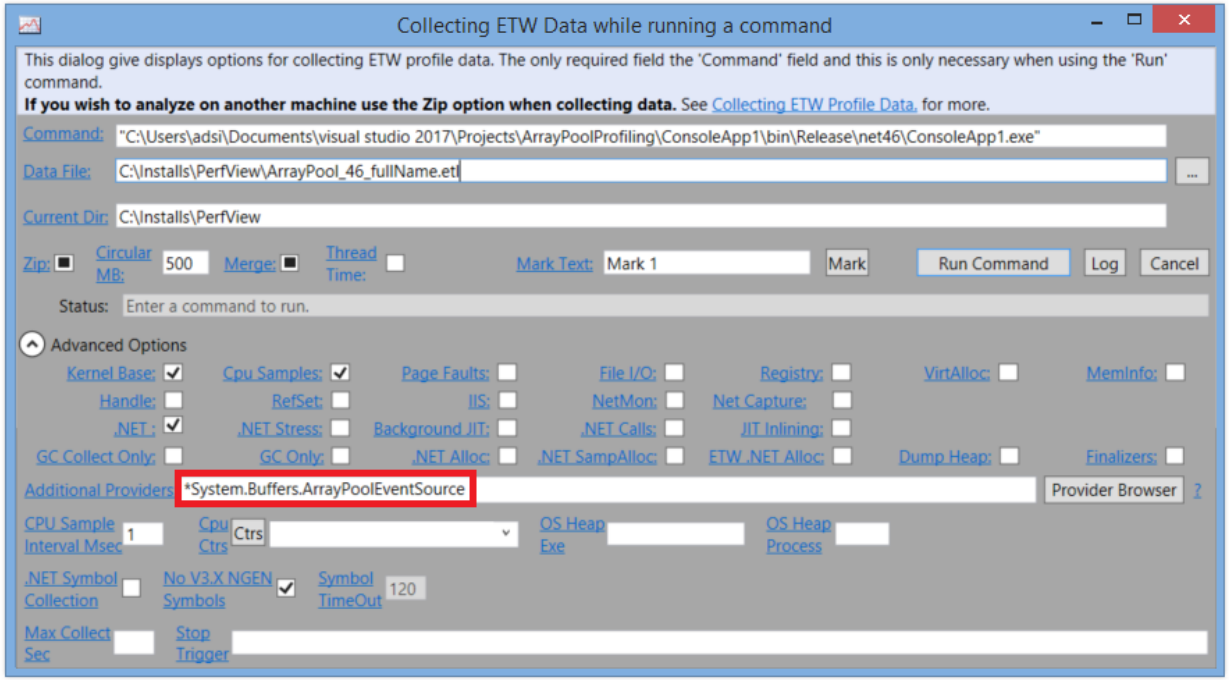
Allocate? Pool?

Method	Size	Mean	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Allocate	100	8.149 ns	0.0169 ns	0.0610	-	-	128 B
RentAndReturn_Shared	100	**43.446 ns**	0.0908 ns	-	-	-	**0 B**
Allocate	1 000	41.122 ns	0.0812 ns	0.4880	0.0000	-	1024 B
RentAndReturn_Shared	1 000	**42.535 ns**	0.0621 ns	-	-	-	**0 B**
Allocate	10 000	371.113 ns	3.2994 ns	4.7847	0.0000	-	10024 B
RentAndReturn_Shared	10 000	**42.565 ns**	0.0450 ns	-	-	-	**0 B**
Allocate	100 000	3,625.029 ns	17.2533 ns	31.2497	31.2497	31.2497	100024 B
RentAndReturn_Shared	100 000	**42.426 ns**	0.0555 ns	-	-	-	**0 B**
Allocate	1 000 000	18,769.792 ns	60.4307 ns	249.9980	249.9980	249.9980	1000024 B
RentAndReturn_Shared	1 000 000	**41.979 ns**	0.0555 ns	-	-	-	**0 B**

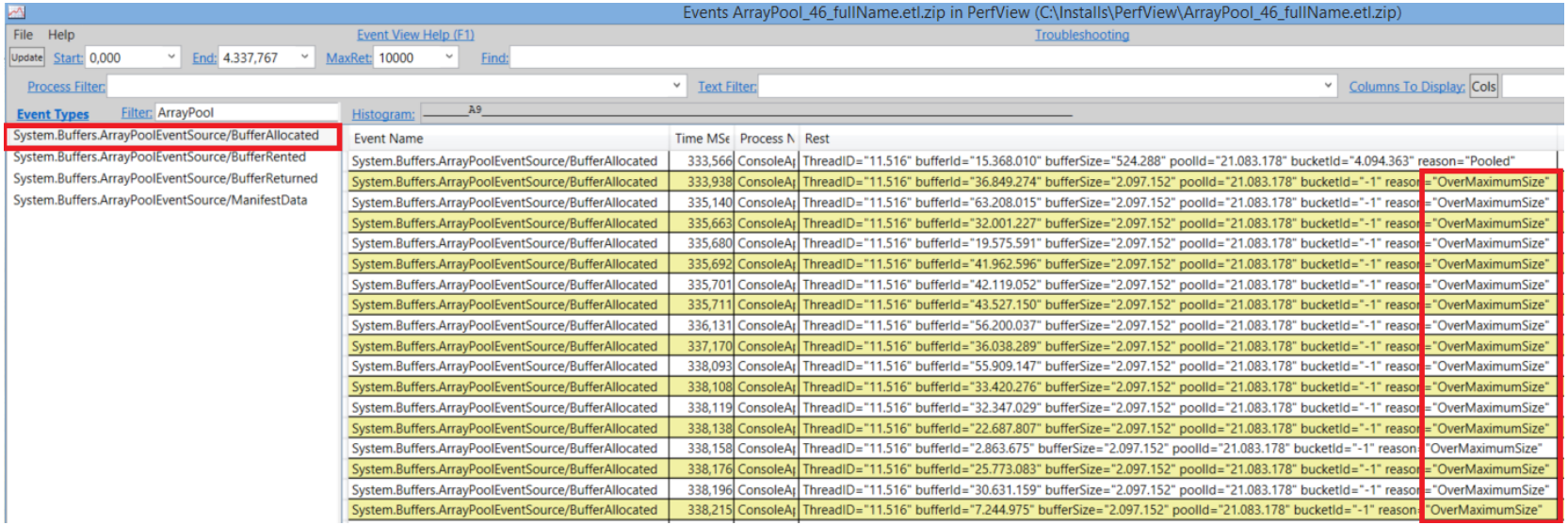
Allocate? Pool?

Method	Size	Mean	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Allocate	100	8.149 ns	0.0169 ns	0.0610	-	-	128 B
RentAndReturn_Shared	100	**43.446 ns**	0.0908 ns	-	-	-	**0 B**
Allocate	1 000	41.122 ns	0.0812 ns	0.4880	0.0000	-	1024 B
RentAndReturn_Shared	1 000	**42.535 ns**	0.0621 ns	-	-	-	**0 B**
Allocate	10 000	371.113 ns	3.2994 ns	4.7847	0.0000	-	10024 B
RentAndReturn_Shared	10 000	**42.565 ns**	0.0450 ns	-	-	-	**0 B**
Allocate	100 000	3,625.029 ns	17.2533 ns	31.2497	31.2497	31.2497	100024 B
RentAndReturn_Shared	100 000	**42.426 ns**	0.0555 ns	-	-	-	**0 B**
Allocate	1 000 000	18,769.792 ns	60.4307 ns	249.9980	249.9980	249.9980	1000024 B
RentAndReturn_Shared	1 000 000	**41.979 ns**	0.0555 ns	-	-	-	**0 B**
Allocate	10 000 000	521,016.536 ns	55,326.9203 ns	211.2695	211.2695	211.2695	10000024 B
RentAndReturn_Shared	10 000 000	639,916.968 ns	116,288.7309 ns	206.3623	206.3623	206.3623	10000024 B
RentAndReturn_**Aware**	10 000 000	**47.200 ns**	0.0407 ns	-	-	-	**0 B**

System.Buffers.ArrayPoolEventSource



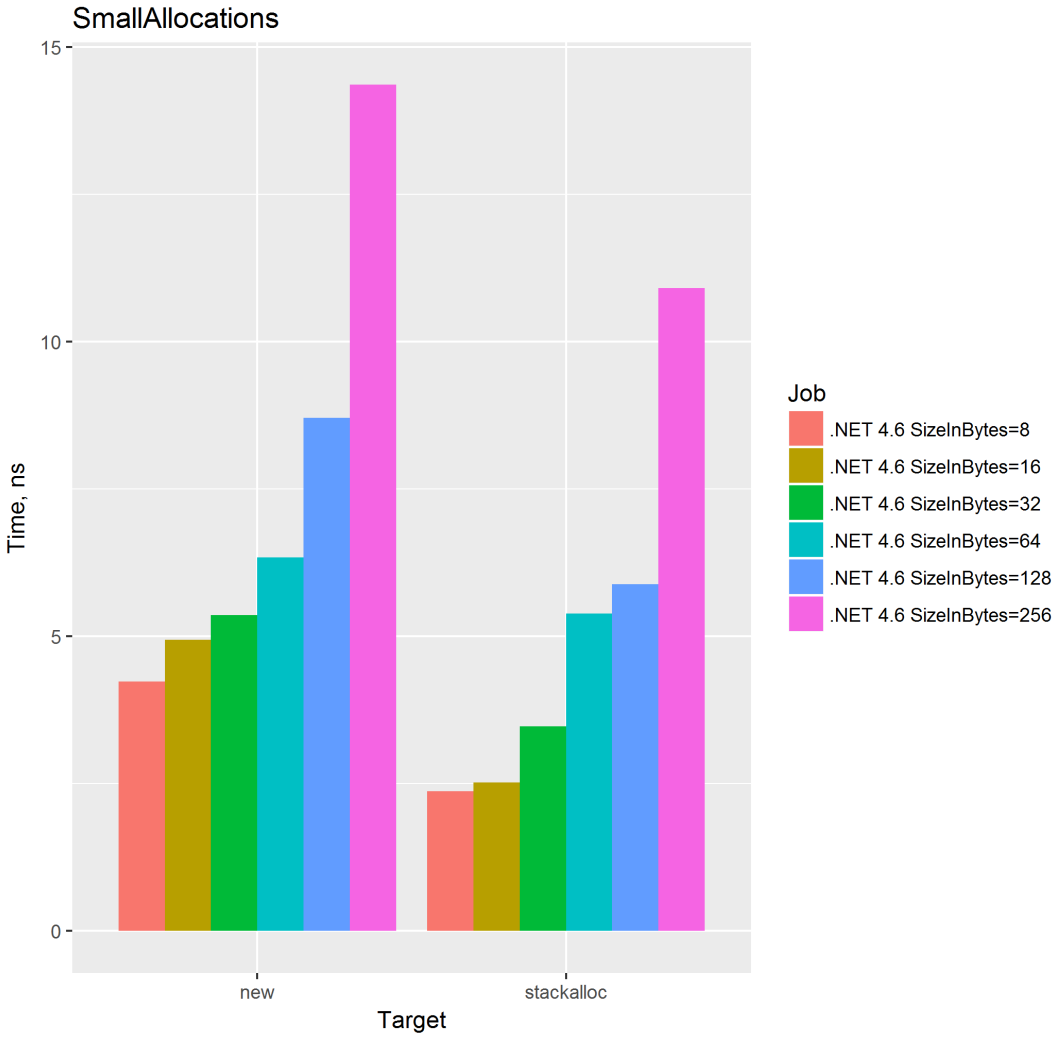
BufferAllocated event



ArrayPool: Summary

- **LOH = Gen 2 = Full GC**
- ArrayPool was designed for best possible performance
- Pool the memory if you can control the lifetime
- Use **Pool.Shared** by default
- Pool allocates the memory for buffers > maxSize
- **The fewer pools, the smaller LOH, the better!**

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



BenchmarkDotNet v0.10.7

	Allocation	Deallocation	Usage
Managed < 85 KB	Very fast	• Non-deterministic • Blocking	• Very easy • Common • Safe
Managed: LOH	Fast	• Very slow	
Stackalloc	Super fast	• Deterministic • Super fast	• Unsafe • Not common • Limited
Marshal	Fast	• Deterministic • Fast	

APIs before Span: parsing integer

```
int Parse(string input);

int Parse(string input, int startIndex, int length);

int Parse(string input, long startIndex, int length);

unsafe int Parse(char* input, int length);

unsafe int Parse(char* input, long startIndex, int length);
```

Span<T>

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 stack only Value Type.

Supports **any** 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[] { 'i', 'm', 'p', 'l', 'i', 'c', 'i', 't' };
Span<char> fromArray = array; // implicit cast


ReadOnlySpan<char> fromString = "State of the .NET Performance".AsSpan();
```

Simple API*

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

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

public void Clear();
public void Fill(T value);

public void CopyTo(Span<T> destination);
public bool TryCopyTo(Span<T> destination);

public ref T DangerousGetPinnableReference();
```

* It's not the full list

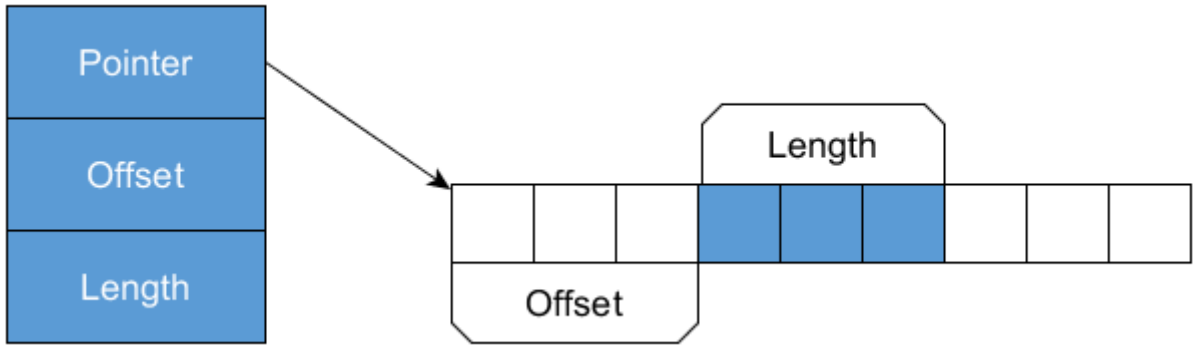
API Simplicity!

```
int Parse(Span<char> input)

void Copy<T>(Span<T> source, Span<T> destination)
```

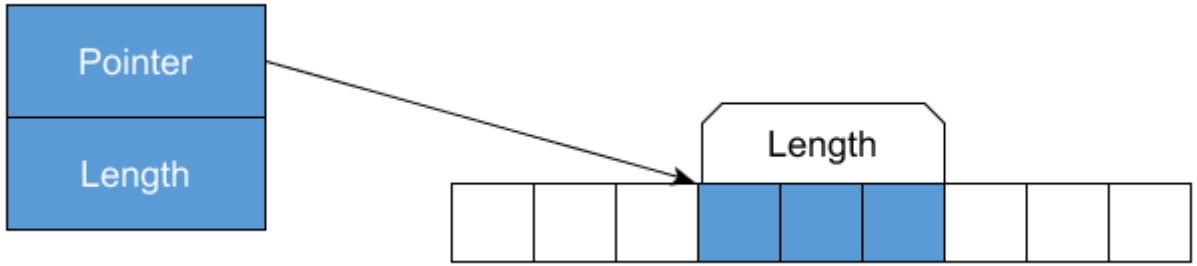
Span for existing runtimes

.NET Standard 1.0 (.NET 4.5+)



Span for new runtimes

.NET Core 2.0 and any other runtime supporting by-ref fields



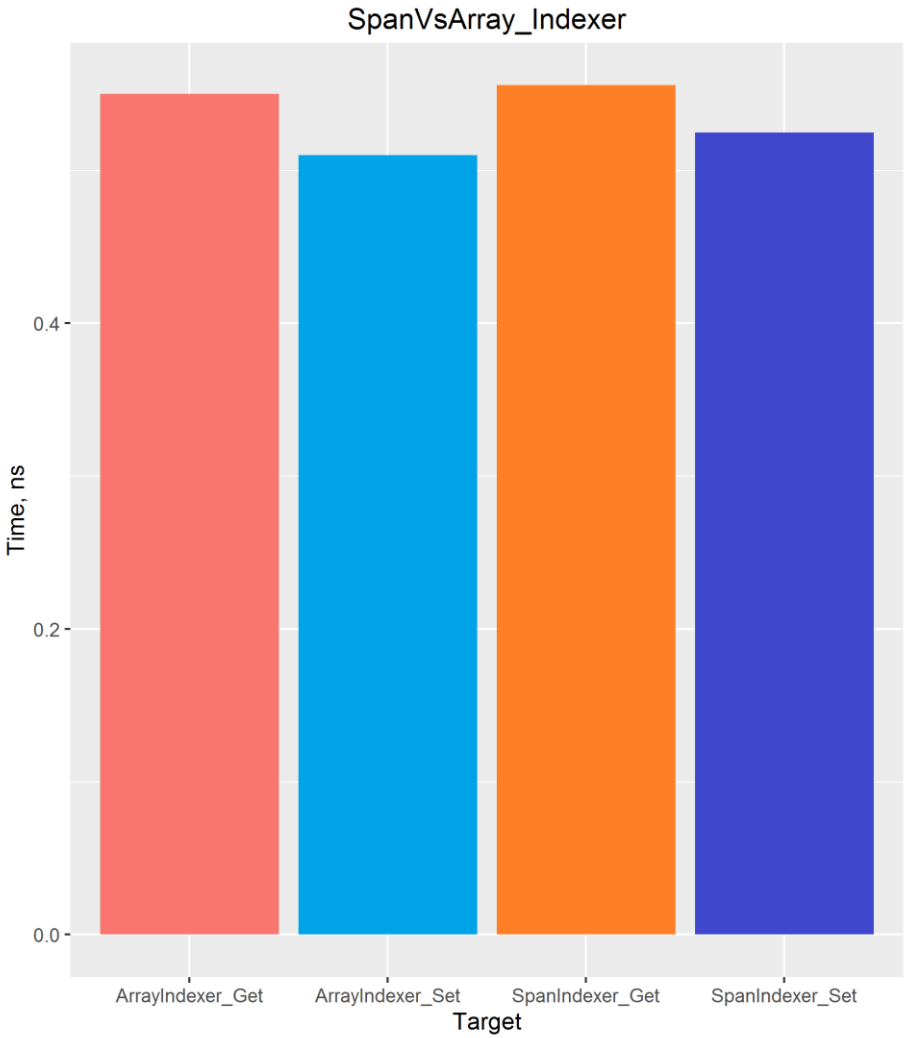
“Fast” vs “Slow” Span

Method	Job	Mean	Scaled
SpanIndexer_Get	.NET 4.6	0.6119 ns	1.14
SpanIndexer_Get	.NET Core 1.1	0.6092 ns	1.13
SpanIndexer_Get	**.NET Core 2.0**	**0.5368 ns**	**1.00**
SpanIndexer_Set	.NET 4.6	0.6117 ns	1.13
SpanIndexer_Set	.NET Core 1.1	0.6082 ns	1.12
SpanIndexer_Set	**.NET Core 2.0**	**0.5417 ns**	**1.00**

There is some place for further improvement!

Span is on par with Array*!

*.NET Core 2.0



BenchmarkDotNet v0.10.6.202-nightly

Creating substrings **before Span** (pseudocode)

```
string Substring(string text, int startIndex, int length)
{
    string result = new string(length); // ALLOCATION!

    Memory.Copy(text, result, startIndex, length); // COPYING

    return result;
}
```

Creating substrings **without allocation**! (pseudocode)

```
ReadOnlySpan<char> Slice(string text, int startIndex, int length)
    => new ReadOnlySpan<char>(
            ref text[0] + (startIndex * sizeof(char)),
            length);
```

Substring vs Slice

Method	Chars	Mean	StdDev	Scaled	Gen 0	Allocated
Substring	10	**8.277** ns	0.1938 ns	**4.54**	**0.0191**	**40 B**
Slice	10	**1.822** ns	**0.0383** ns	1.00	**-**	0 B
Substring	1000	**85.518** ns	1.3474 ns	**47.22**	**0.4919**	**1032 B**
Slice	1000	**1.811** ns	**0.0205** ns	1.00	**-**	0 B

Possible usages

- Parsing without allocations
- Formatting
- Base64/Unicode encoding
- HTTP Parsing/Writing
- Compression/Decompression
- XML/JSON parsing/writing
- Binary reading/writing
- & more!!

Stack Only

- Instances can reside only on the stack
- Which is accessed by one thread at the same time

Advantages:

- Few pointers for GC to track
- Safe Concurrency (no Struct Tearing)
- Safe lifetime. Method ends = memory can be returned to the pool or released

The Limitations http://adamsitnik.com/Span/#the-limitations

Span: Summary

- Allows to work with **any** type of memory.
- It makes working with native memory much easier.
- Simple abstraction over Pointer Arithmetic.
- **Avoid allocation and copying of memory with Slicing.**
- Supports .NET Standard 1.0+
- It’s perfromance is on par with Array for new runtimes.
- It’s limited due to stack only requirements.

System.Runtime.CompilerServices.Unsafe

Overcoming C# limitations:

- Managed Pointer Arithmetic
- Casting w/o constraints
- Copy/Init Block
- Read/Write w/o constraints
- SizeOf(T)

```
ref T AddByteOffset<T>(ref T source, IntPtr byteOffset)
ref T Add<T>(ref T source, int elementOffset)
ref T Add<T>(ref T source, IntPtr elementOffset)
bool AreSame<T>(ref T left, ref T right)
void* AsPointer<T>(ref T value)
ref T AsRef<T>(void* source)
T As<T>(object o) where T : class
ref TTo As<TFrom, TTo>(ref TFrom source)
IntPtr ByteOffset<T>(ref T origin, ref T target)
void CopyBlock(ref byte destination, ref byte source, uint byteCount)
void CopyBlock(void* destination, void* source, uint byteCount)
void CopyBlockUnaligned(ref byte destination, ref byte source, uint byteCount)
void CopyBlockUnaligned(void* destination, void* source, uint byteCount)
void Copy<T>(void* destination, ref T source)
void Copy<T>(ref T destination, void* source)
void InitBlock(ref byte startAddress, byte value, uint byteCount)
void InitBlock(void* startAddress, byte value, uint byteCount)
void InitBlockUnaligned(ref byte startAddress, byte value, uint byteCount)
void InitBlockUnaligned(void* startAddress, byte value, uint byteCount)
T Read<T>(void* source)
T ReadUnaligned<T>(void* source)
T ReadUnaligned<T>(ref byte source)
int SizeOf<T>()
ref T SubtractByteOffset<T>(ref T source, IntPtr byteOffset)
ref T Subtract<T>(ref T source, int elementOffset)
ref T Subtract<T>(ref T source, IntPtr elementOffset)
void Write<T>(void* destination, T value)
void WriteUnaligned<T>(void* destination, T value)
void WriteUnaligned<T>(ref byte destination, T value)
```

.NET Standard

Package name	.NET Standard	.NET Framework
System.Memory	1.0	4.5
System.Buffers	1.1	4.5.1
System.Threading.Tasks.Extensions	1.0	4.5
System.Runtime.CompilerServices.Unsafe	1.0	4.5

Summary

- Use ValueTask only if it can help you!
- Pool the memory with ArrayPool
- Use Span and slicing to avoid allocations
- Use Span to take advantage of the native memory
- Use the “Unsafe” api to use C# only

Sources

- Span<T> design document
- Compile time enforcement of safety for ref-like types
- ValueTask doesn't inline well- GitHub issue

Děkuji!

Slides: http://adamsitnik.com/files/Prague.pdf

Code: https://github.com/adamsitnik/StateOfTheDotNetPerformance

@SitnikAdam
Adam.Sitnik@gmail.com