

State of the .NET Performance

Adam Sitnik

About myself

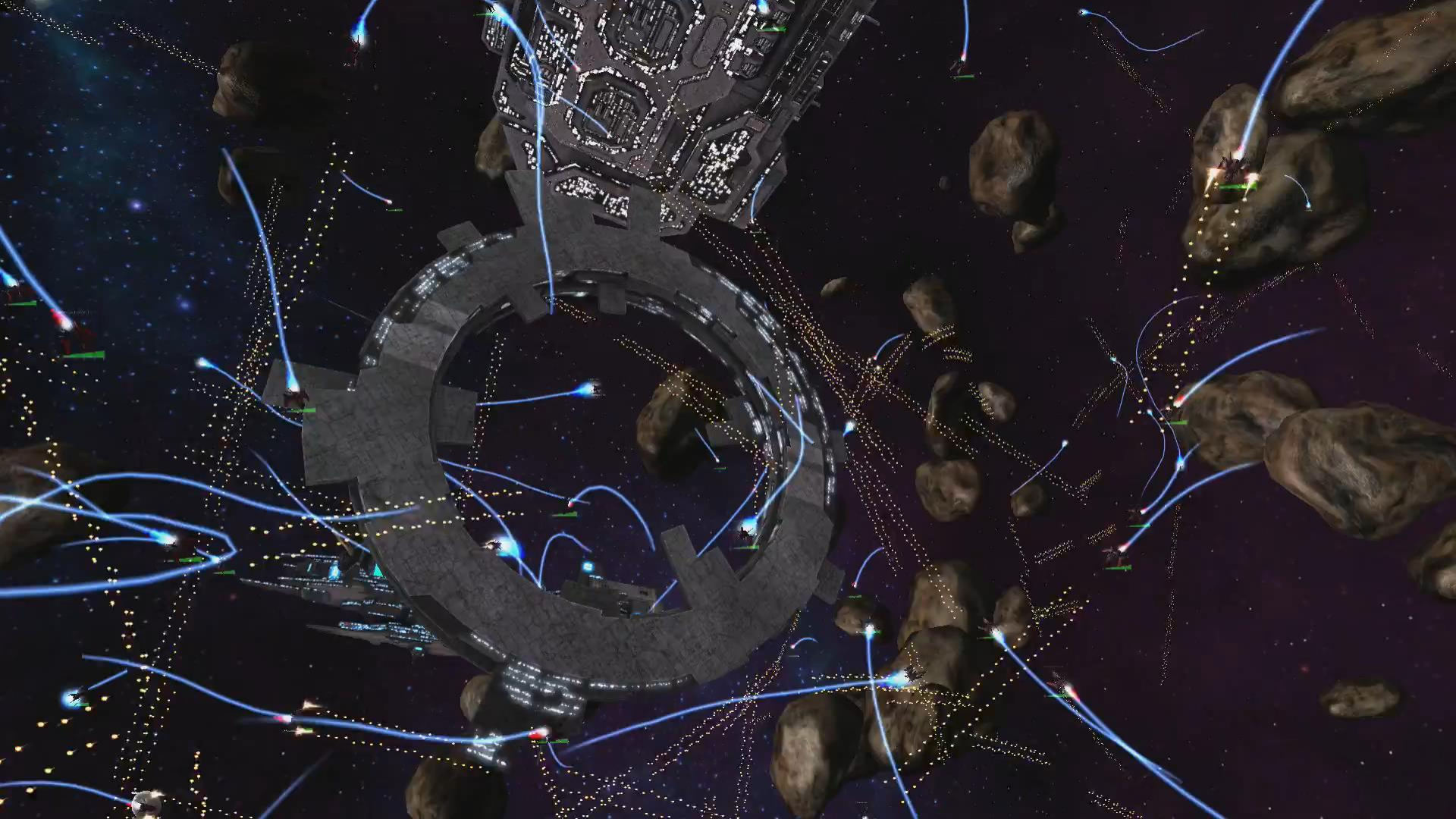
Open Source:

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

Work:



- Energy Trading
(.NET Core running in production since July 2016)



How to get the best of GC?

- Use the right GC mode
- Reduce allocations
- Eliminate all managed allocations:
 - Use Value Types
 - Pool the managed memory
 - Use unmanaged memory

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);
}
```

Value Tuple vs Tuple

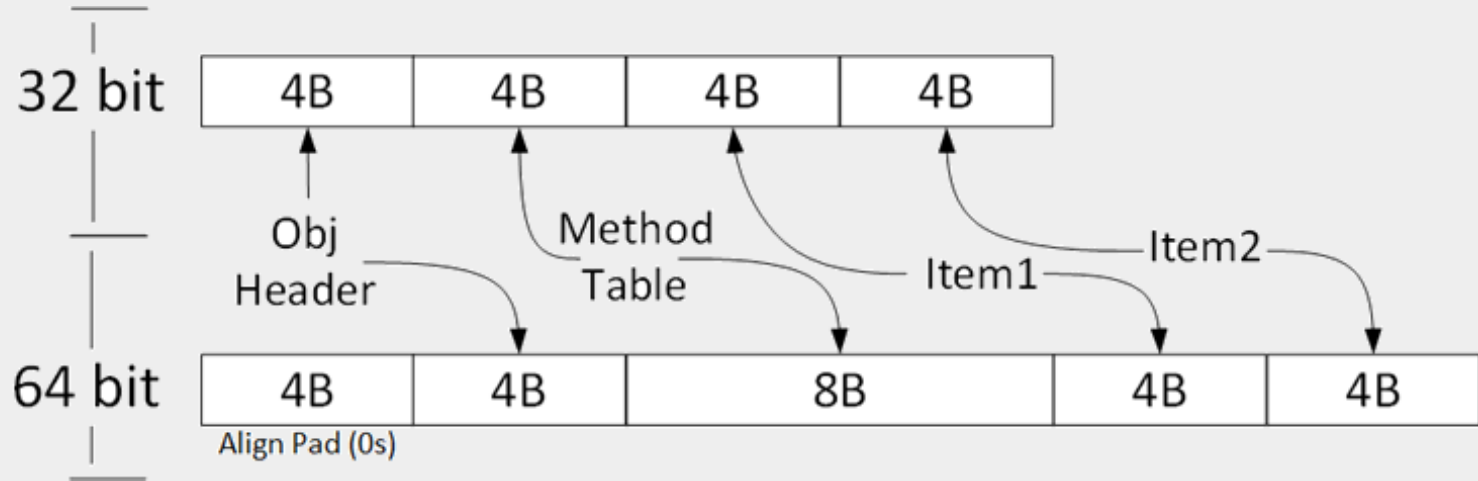
```
struct ValueTuple<T1, T2>  
{  
    T1 Item1;  
    T2 Item2;  
}
```

```
class Tuple<T1, T2>  
{  
    T1 Item1;  
    T2 Item2;  
}
```

In the following examples we will consider
(Value)Tuple<int, int>

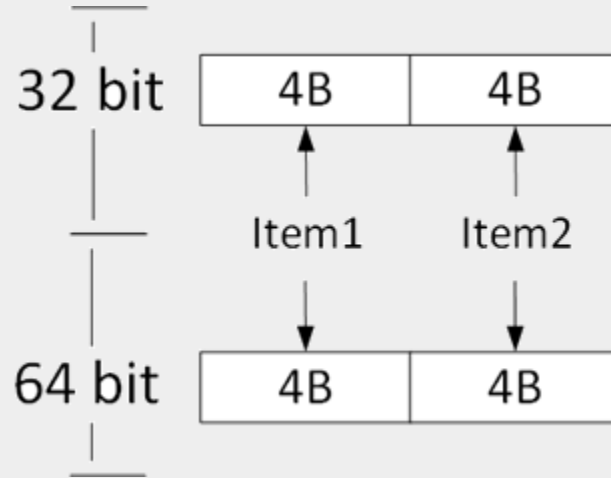
Reference Types: Memory Layout

For `Tuple<int, int>`

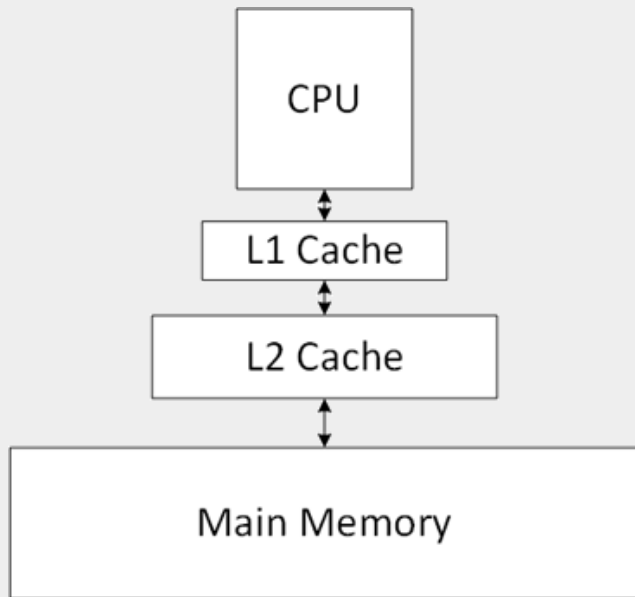


Value Types: Memory Layout

For `ValueTuple<int, int>`



Caching

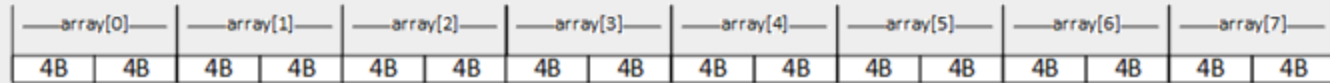
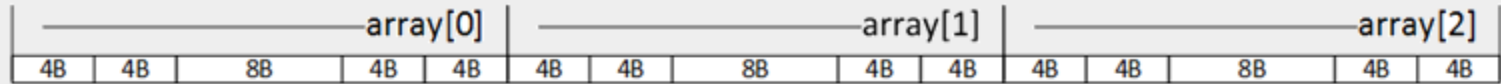


Latency Numbers Every Programmer Should Know

L1 cache reference	0.5 ns
L2 cache reference	7 ns
Main memory reference	100 ns
Disk seek	10,000,000 ns

Less Space = Better Data Locality

Read array[0] => Cache Miss => copy whole Cache Line



Data Locality: Simple Benchmark

```
[Benchmark]
[HardwareCounters(HardwareCounter.CacheMisses)]
public int Iterate()
{
    int item1Sum = 0, item2Sum = 0;
    var array = arrayOfGivenType;
    for (int i = 0; i < array.Length; i++)
    {
        item1Sum += array[i].Item1;
        item2Sum += array[i].Item2;
    }
    return item1Sum + item2Sum;
}
```

Types	Scaled	Cache Misses
Value	1.00	4 463 155
Reference	3.22	13 992 618

For array of 100 000 000 elements

Value Types: No GC

```
public (int,int) ValueTuple() => ValueTuple.Create(0, 0);
```

```
public Tuple<int, int> Tuple() => Tuple.Create(0, 0);
```

Type	Platform	Gen 0	Allocated
CreateValueTuple	X86	-	0 B
CreateTuple	X86	0.0051	16 B
CreateValueTuple	X64	-	0 B
CreateTuple	X64	0.0076	24 B

+ **deterministic** deallocation for stack-allocated Value Types

Value Types: the disadvantages?!

- Can be easily boxed
- By default send to and returned from methods by **copy**.
- It's **expensive** to copy non-primitive value types!
- It's not obvious when the copying when it happens!

Defensive copy

```
struct Test {  
    readonly int Field;  
    Test(int value) => Field = value;  
  
    void Ugly(int newValue) => this = new Test(newValue);  
}
```

```
readonly Test field = new Test(1);  
void DemoField() {  
    WriteLine(field.Field);  
    field.Ugly(2);  
    WriteLine(field.Field);  
}
```

1	1
2	1

C# 7.2: readonly types

```
readonly struct Test
{
    public readonly int Field;

    public Test(int value) => Field = value;

    public void Ugly(int newValue)
        => this = new Test(newValue); // compilation error!
}
```

„this“ is readonly reference for readonly types

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; }
```

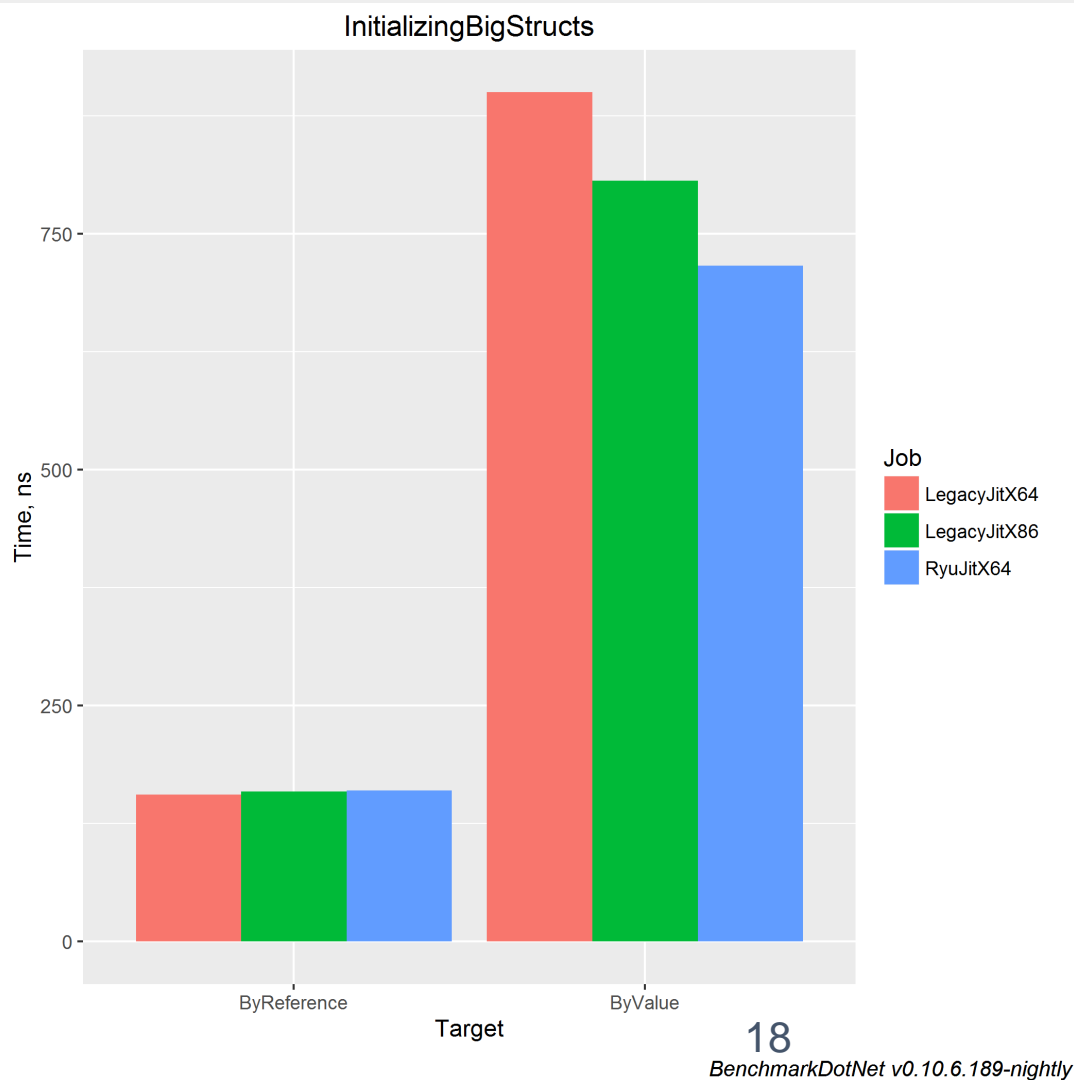
ByValue

```
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;  
}
```

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;  
}
```

How can old JITs support it?



What about unsafe?!

```
public unsafe void ByReferenceUnsafe()
{
    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	6.958 us	4.57
ByReference	RyuJit	1.524 us	1.00
ByReferenceUnsafe	RyuJit	1.540 us	1.01

No need for pinning!

Executing Unsafe code requires **full trust**. It can be a „no go“!

mustoverride.com



Vladimir Sadov

Engineer

 Website

 Twitter

 Github

Posts by Tags

refs

- November 30, 2016 » Why ref locals allow only a single binding?
- November 04, 2016 » Safe to return rules for ref returns.
- October 29, 2016 » Local variables cannot be returned by reference.
- September 17, 2016 » Managed pointers.
- September 05, 2016 » ref returns are not pointers.

tuples

- February 11, 2017 » C# Tuples. Conversions.
- January 28, 2017 » C# Tuples. More about element names.
- January 16, 2017 » C# Tuples. How tuples are related to ValueTuple.
- January 07, 2017 » C# Tuples. Why mutable structs?

C# 7(.2): Performance Summary

- Value Types have better performance characteristics:
 - Data Locality
 - No GC
- Value Tuples offer clean coding and great performance
- Safe references can make your code faster than unsafe!
 - Use them only when needed to keep your code clean
- Using readonly structs can prevent from defensive copying

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()) // INLINE
        : 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;
}
```

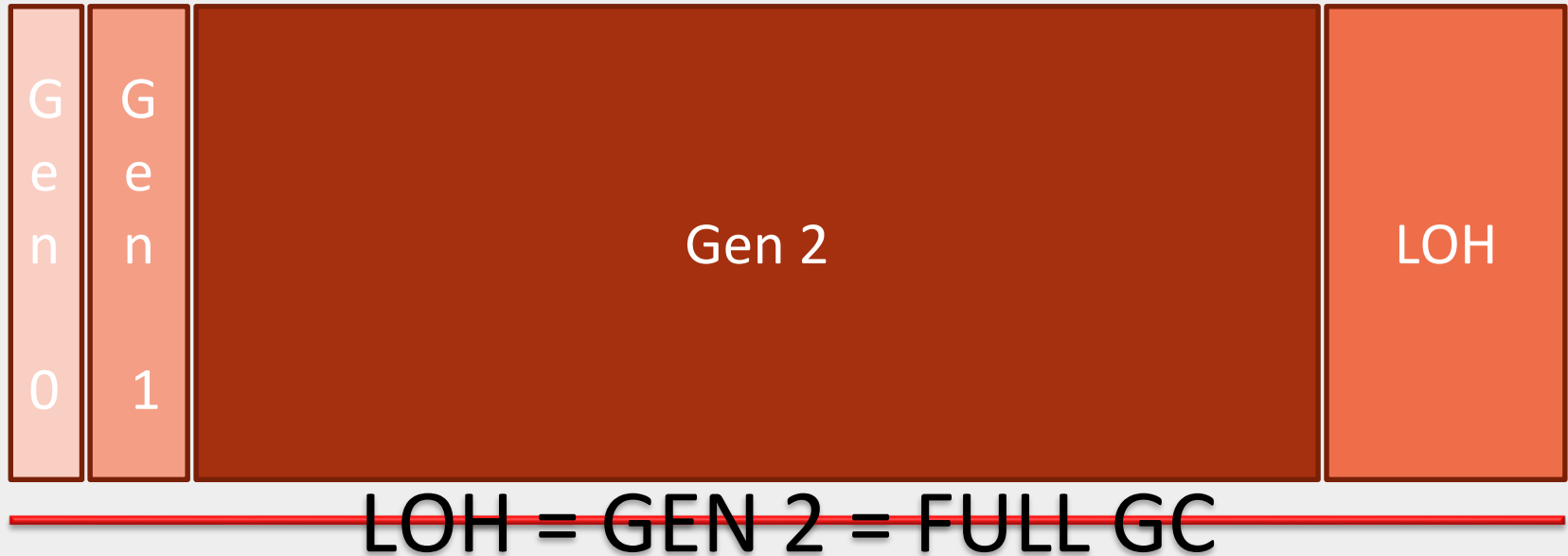
ValueTask vs Task: Overhead Only

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

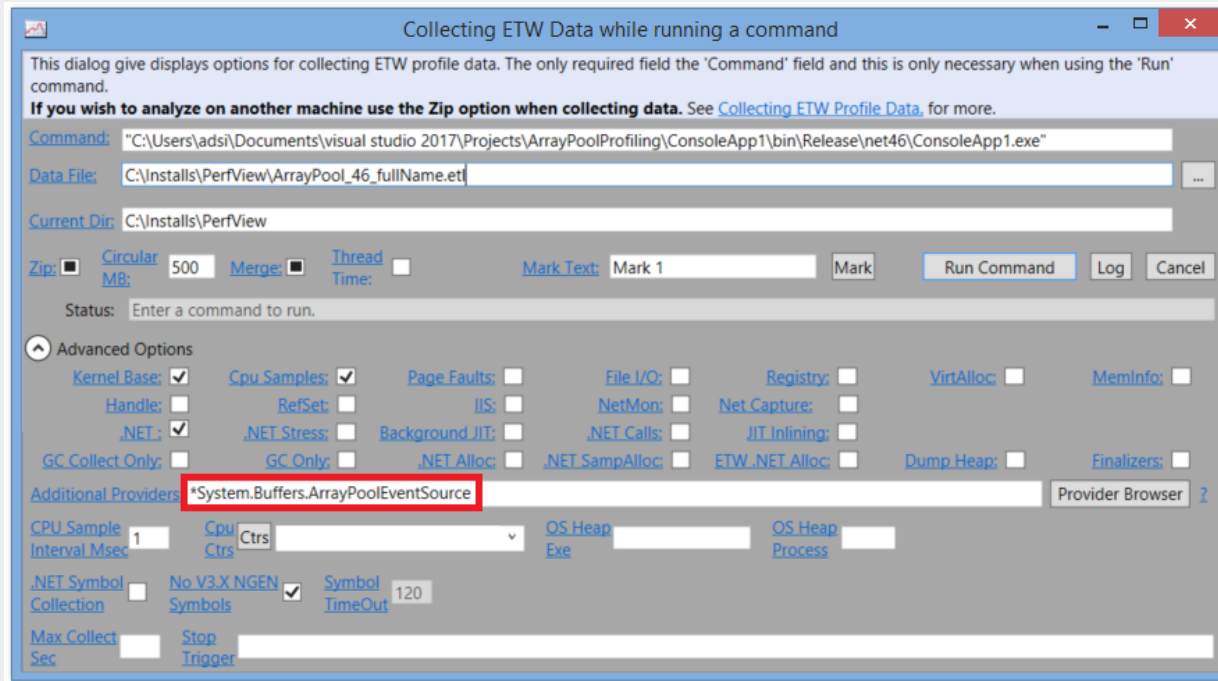
Allocate? Pool?

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

Events ArrayPool_46_fullName.etl.zip in PerfView (C:\Installs\PerfView\ArrayPool_46_fullName.etl.zip)

File Help Event View Help (F1) Troubleshooting

Update Start: 0,000 End: 4,337,767 MaxRet: 10000 Find:

Process Filter: Text Filter: Columns To Display: Cols

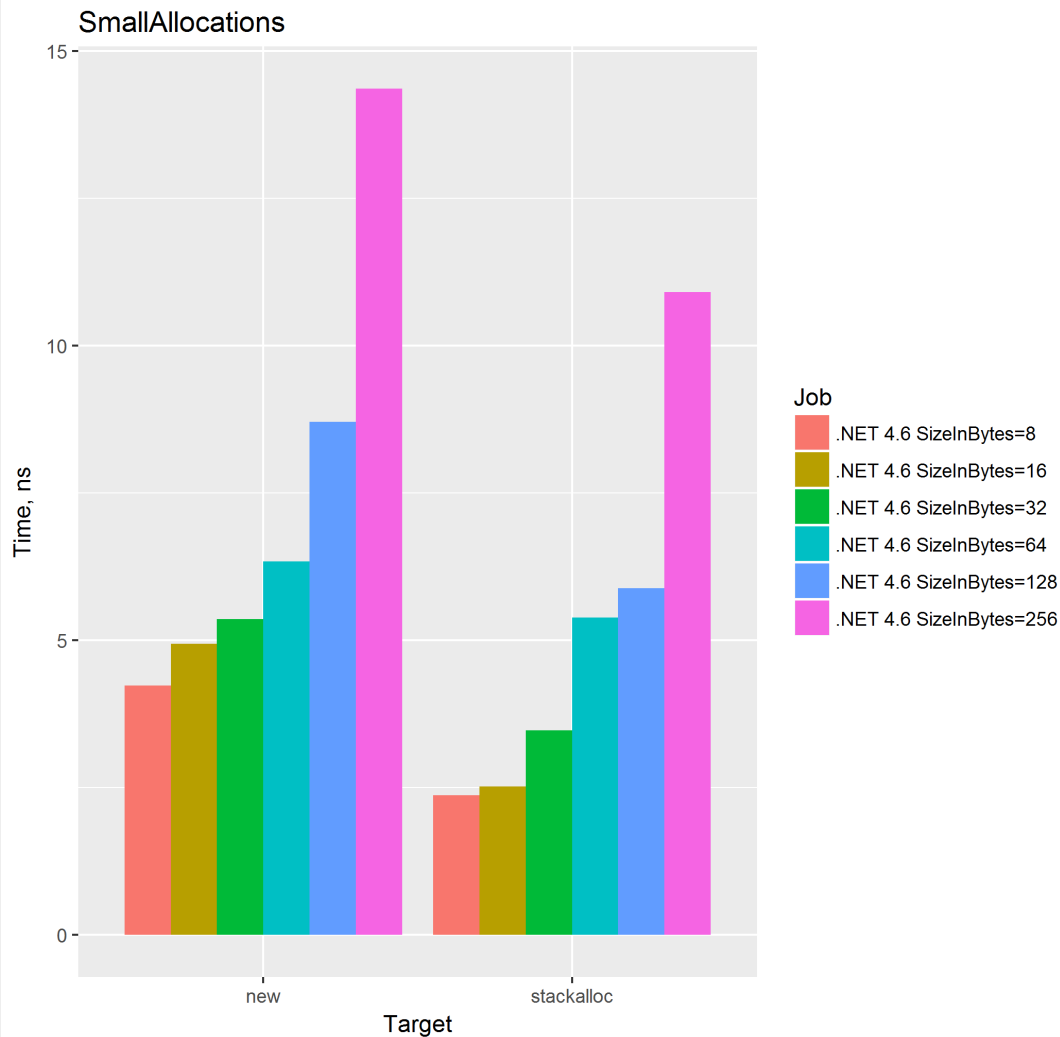
Event Types Filter: ArrayPool Histogram: A9

Event Name	Time MS	Process N	Rest
System.Buffers.ArrayPoolEventSource/BufferAllocated	333,566	ConsoleA	ThreadId="11.516" bufferId="15.368.010" bufferSize="524.288" poolId="21.083.178" bucketId="4.094.363" reason="Pooled"
System.Buffers.ArrayPoolEventSource/BufferAllocated	333,938	ConsoleA	ThreadId="11.516" bufferId="36.849.274" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	335,140	ConsoleA	ThreadId="11.516" bufferId="63.208.015" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	335,663	ConsoleA	ThreadId="11.516" bufferId="32.001.227" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	335,680	ConsoleA	ThreadId="11.516" bufferId="19.575.591" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	335,692	ConsoleA	ThreadId="11.516" bufferId="41.962.596" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	335,701	ConsoleA	ThreadId="11.516" bufferId="42.119.052" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	335,711	ConsoleA	ThreadId="11.516" bufferId="43.527.150" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	336,131	ConsoleA	ThreadId="11.516" bufferId="56.200.037" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	337,170	ConsoleA	ThreadId="11.516" bufferId="36.038.289" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,093	ConsoleA	ThreadId="11.516" bufferId="55.909.147" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,108	ConsoleA	ThreadId="11.516" bufferId="33.420.276" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,119	ConsoleA	ThreadId="11.516" bufferId="32.347.029" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,138	ConsoleA	ThreadId="11.516" bufferId="22.687.807" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,158	ConsoleA	ThreadId="11.516" bufferId="2.863.675" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,176	ConsoleA	ThreadId="11.516" bufferId="25.773.083" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,196	ConsoleA	ThreadId="11.516" bufferId="30.631.159" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"
System.Buffers.ArrayPoolEventSource/BufferAllocated	338,215	ConsoleA	ThreadId="11.516" bufferId="7.244.975" bufferSize="2.097.152" poolId="21.083.178" bucketId="-1" reason="OverMaximumSize"

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



	Allocation	Deallocation	Usage
Managed < 85 KB	Very fast	<ul style="list-style-type: none"> • Non-deterministic • Blocking • Very slow 	<ul style="list-style-type: none"> • Very easy • Common • Safe
Managed: LOH	Fast		
Stackalloc	Super fast	<ul style="list-style-type: none"> • Deterministic • Super fast 	<ul style="list-style-type: none"> • Unsafe • Not common • Limited
Marshal	Fast	<ul style="list-style-type: none"> • 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 **read only** and **stack only Value Type**.

Supports **any** memory

```
Span<byte> stackMemory = stackalloc byte[256]; // C# 7.2
```

```
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

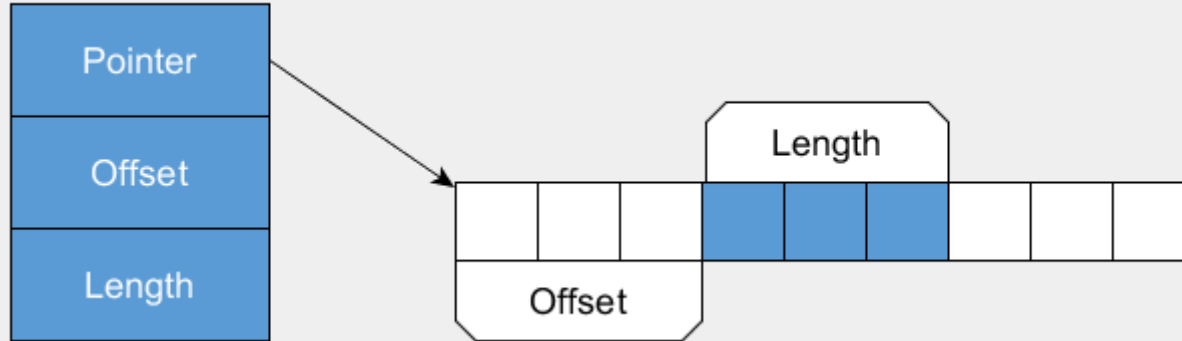
API Simplicity!

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

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

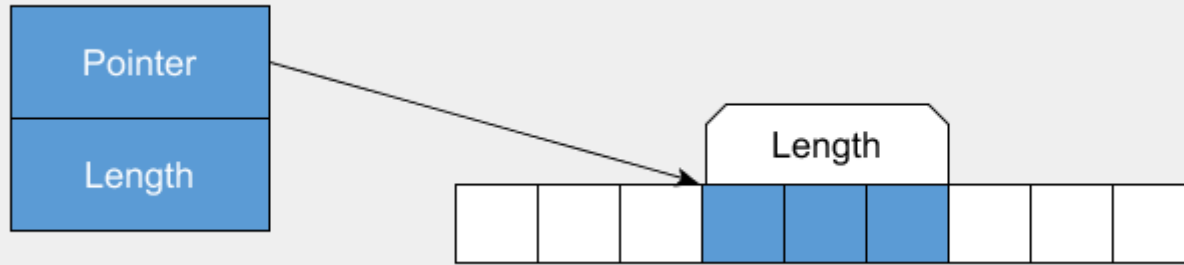

Span for existing runtimes

.NET Standard 1.0 (.NET 4.5+)



Span for new runtimes

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



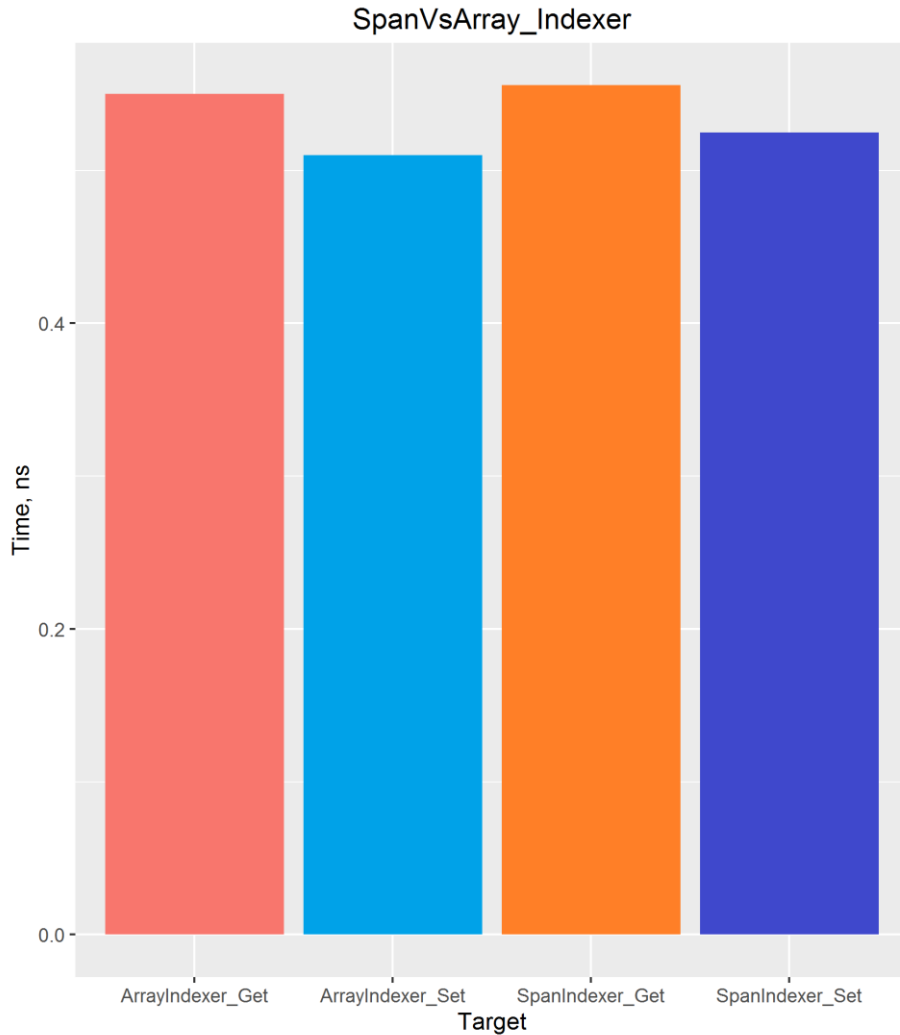
“Fast” vs “Slow” Span

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



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

Stack Only: No Heap Limitations

```
void NonConstrained<T>(IEnumerable<T> collection)

struct SomeValueType<T> : IEnumerable<T> { }

void Demo()
{
    var value = new SomeValueType<int>();
    NonConstrained(value);
}
```

Boxing == Heap. Heap != Stack

```
.method private hidebysig
instance void Demo () cil managed
{
    // Method begins at RVA 0x2054
    // Code size 21 (0x15)
    .maxstack 2
    .locals init (
        [0] valuetype Sample.SomeValueType`1<int32> 'value'
    )

    IL_0000: ldloc.s 'value'
    IL_0002: initobj valuetype Sample.SomeValueType`1<int32>
    IL_0008: ldarg.0
    IL_0009: ldloc.0
    IL_000a: box valuetype Sample.SomeValueType`1<int32>
    IL_000f: call instance void Sample.Program::NonConstrained<int32>(class
    IL_0014: ret
} // end of method Program::Demo
```

Stack Only: Even More Limitations

```
async Task Method(StackOnly<byte> bytes)
```

```
class SomeClass  
{  
    StackOnly<byte> field;  
}
```

```
Func<StackOnly<byte>> genericArgument;
```

Memory<T>

- a type complementing Span<T>
- must not be created for stack memory

```
public Span<T> Span { get; }
```

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

OwnedMemory<T>

```
public abstract class OwnedMemory<T> : IDisposable, IRetainable
{
    public Memory<T> AsMemory { get; }
    public abstract bool IsDisposed { get; }
    protected abstract bool IsRetained { get; }
    public abstract int Length { get; }
    public abstract Span<T> AsSpan();
    public void Dispose();
    protected abstract void Dispose(bool disposing);
    public abstract MemoryHandle Pin();
    public abstract bool Release();
    public abstract void Retain();
    protected internal abstract bool TryGetArray(out ArraySegment<T> arraySegment);
}
```

.NET Standard 2.1: Span based APIs

```
namespace System.IO
{
    public class Stream
    {
        public virtual int Read(Span<byte> destination);
        public virtual ValueTask<int> ReadAsync(Memory<byte> destination);

        public virtual void Write(ReadOnlySpan<byte> source);
        public virtual Task WriteAsync(ReadOnlyMemory<byte> source);
    }
}
```

.NET Standard 2.1: Span based APIs

- System.BitConverter, System.Convert
- System.Random
- System.Int16, System.DateTime, System.DateTimeOffset, System.TimeSpan, System.Version, System.Guid
- System.String, System.Text.StringBuilder, System.Text.Encoding
- System.IO.Stream, System.IO.TextReader, System.IO.TextWriter
- System.IO.BinaryReader, System.IO.BinaryWriter
- System.Numerics
- System.Net.IPAddress, System.Net.Sockets, System.Net.WebSockets, System.Net.Http
- System.Security.Cryptography

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 performance is on par with Array for new runtimes.
- It's limited due to stack only requirements.
- Use Memory/OwnedMemory to overcome Span limitations

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

- Start using Value Types today!
- Use references to avoid copying of Value Types.
- Forget about unsafe, use “ref returns and locals” instead
- 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 Memory/OwnedMemory to overcome Span limitations
- Use the “Unsafe” api to use C# only

Sources

- [Series of Great Blog Posts by Vladimir Sadv](#)
- [Span<T> design document](#)
- [Compile time enforcement of safety for ref-like types](#)
- [Add initial Span/Buffer-based APIs across corefx](#)
- [ValueTask doesn't inline well- GitHub issue](#)

FRAGEN?

Slides: <http://adamsitnik.com/files/Cologne.pdf>

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

Ich freue mich auf Ihr Feedback!

Vielen Dank!

@SitnikAdam

Adam.Sitnik@gmail.com