State of the .NET Performance

Adam Sitnik

About myself

Open Source:

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

Work:

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

Agenda

- C# 7
  - ValueTuple
  - ref returns and locals
- .NET Core
  - Span
  - ArrayPool
  - ValueTask
  - ~~Pipelines~~
  - Unsafe
- .NET Standard
- C# 7.2 – “safe, efficient, low-level code”

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

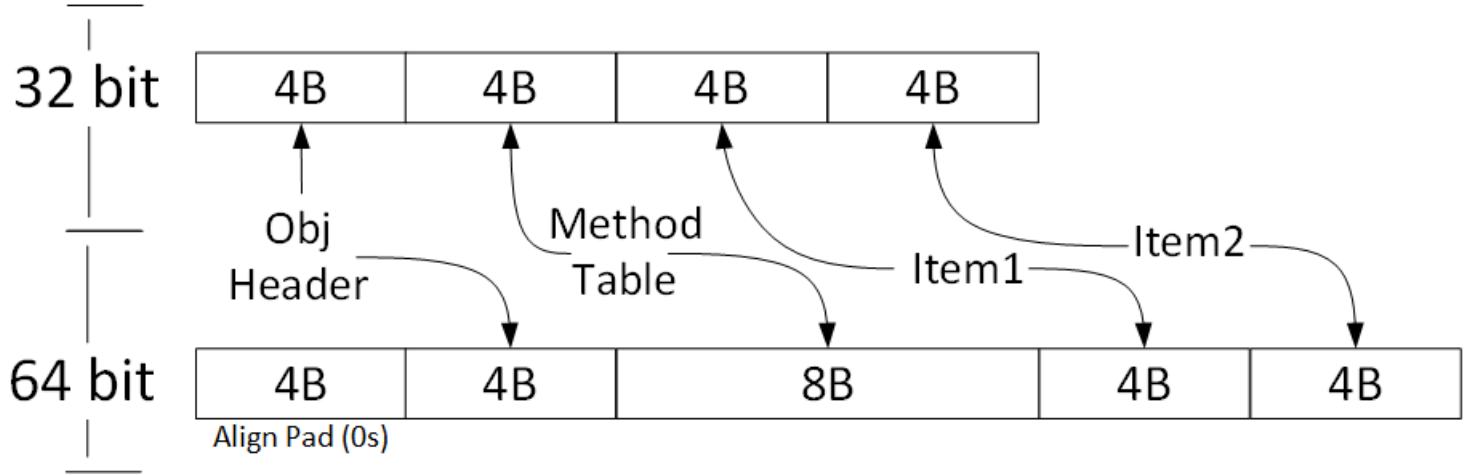
```
struct ValueTuple<T1, T2>
{
    T1 Item1;
    T2 Item2;
}
```

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

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

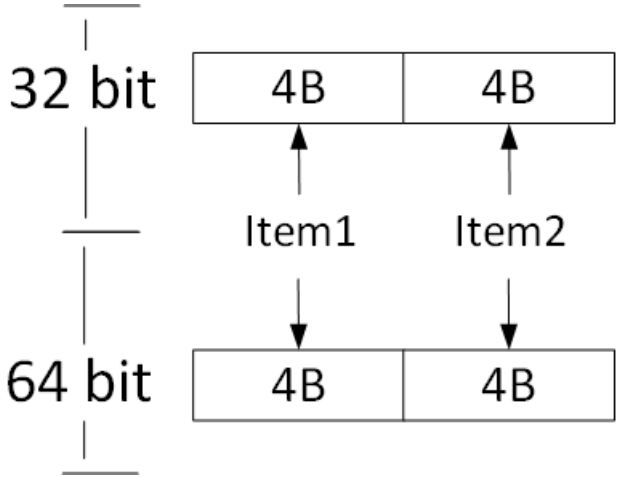
Reference Types: Memory Layout

For `Tuple<int, int>`

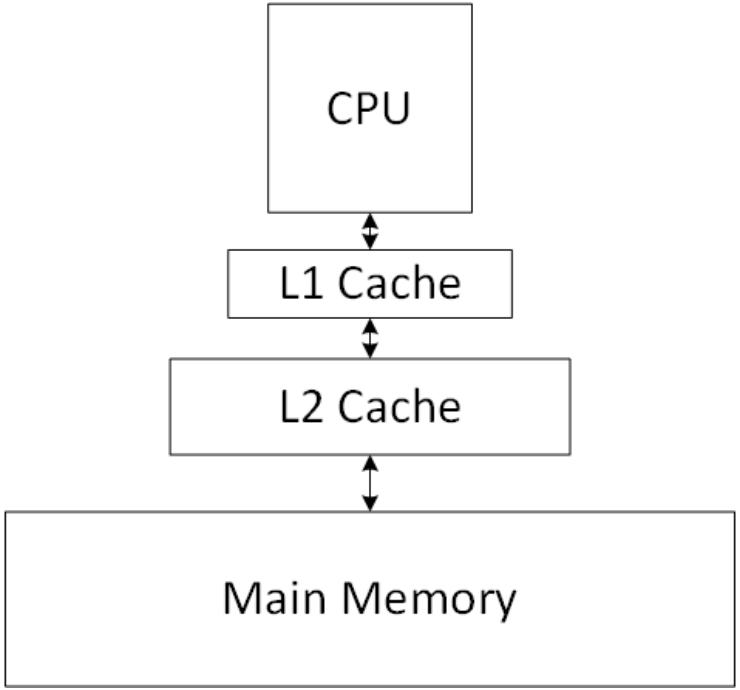


Value Types: Memory Layout

For `ValueTuple<int, int>`



Caching

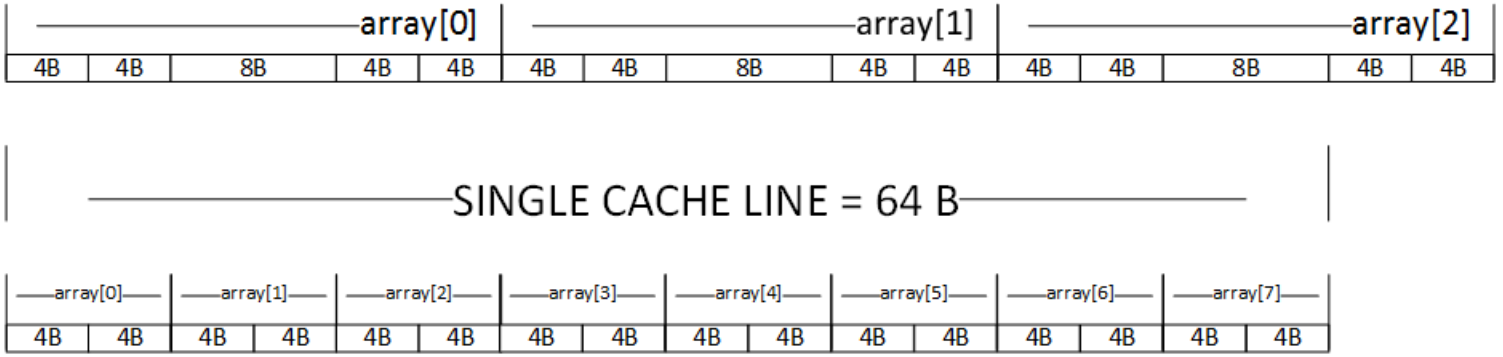


Latency Numbers Every Programmer Should Know

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?!

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

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

is converted to:

```
var copy = readOnlyStruct;
var 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; }
```

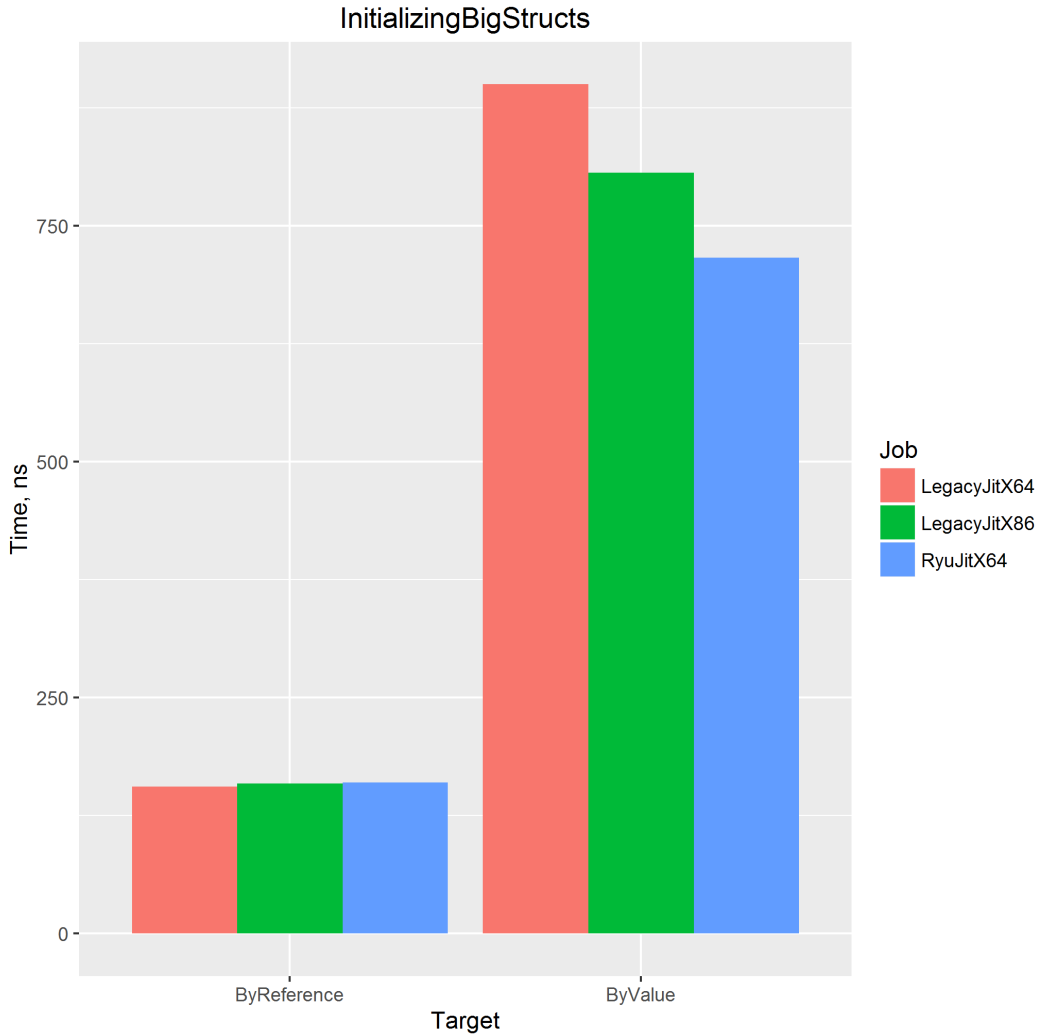
ByValue

```
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?



BenchmarkDotNet v0.10.6.189-nightly

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.

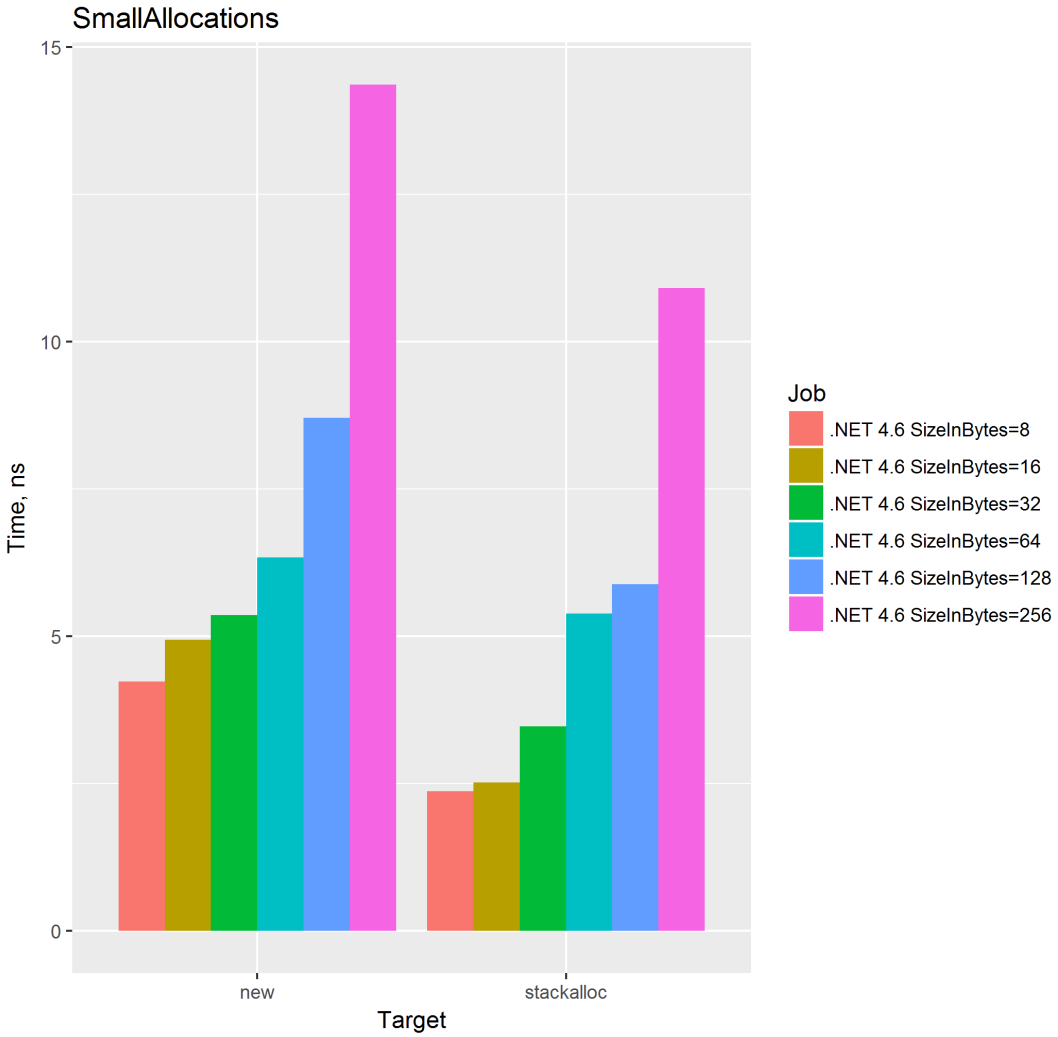
tuples

- 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: 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

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



BenchmarkDotNet v0.10.7

<table>
<tr><th></th><th>Allocation</th><th>Deallocation</th><th>Usage</th></tr>
<tr><td>Managed < 85 KB</td><td>Very cheap (NextObjPtr)</td><td rowspan="2">• non-deterministic
• Expensive!
• GC: stop the world

The same as above &:
• Fragmentation (LOH)
• LOH = Gen 2 = Full GC</td><td rowspan="2">• Very easy
• Common
• Safe</td></tr>
<tr><td>Managed: LOH</td><td>Acceptable cost (free list management)</td></tr>
<tr><td>Stackalloc</td><td>Very cheap</td><td>• Deterministic
• Very cheap</td><td rowspan="2">• Unsafe
• Not common
• Limited</td></tr>
<tr><td>Native: Marshal</td><td>Acceptable cost (free list management)</td><td>• Deterministic
• Very cheap
• On demand</td></tr>
</table>

APIs before Span

```
void Method<T>(T[] input)
void Method<T>(T[] input, int startIndex, int length)
unsafe void Method<T>(void* input, int length)
unsafe void Method<T>(void* input, int startIndex, int length)
void Method<T>(T[] input, T[] output)
void Method<T>(T[] input, int inputStartIndex, int inputLength,
      T[] output, int outputStartIndex, int outputLength)
unsafe void Method<T>(void* input, int inputLength, void* output)
unsafe void Method<T>(void* input, int startIndex, int length,
      void* output, int outputStartIndex, int outputLength)
unsafe void Method<T>(void* input, int inputLength, T[] output)
unsafe void Method<T>(void* input, int startIndex, int length,
      T[] output, int outputStartIndex, int outputLength)
// it's not a full list!
```

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

APIs before Span

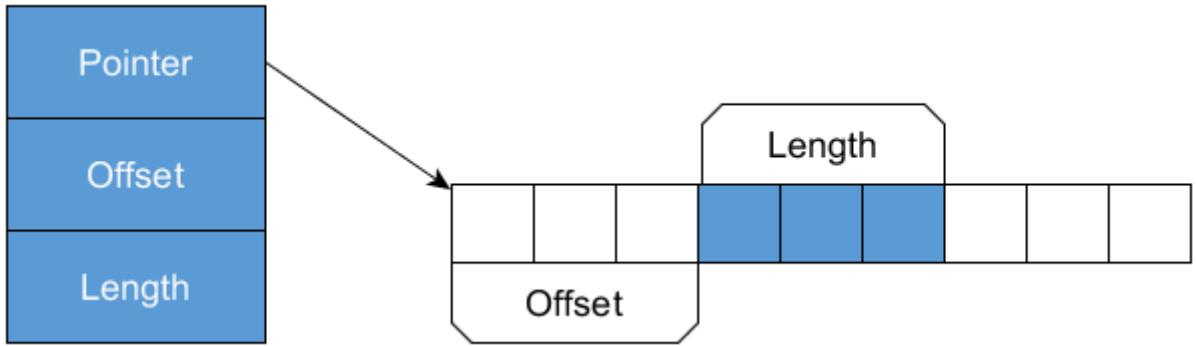
```
void Method<T>(T[] input)
void Method<T>(T[] input, int startIndex, int length)
unsafe void Method<T>(void* input, int length)
unsafe void Method<T>(void* input, int startIndex, int length)
void Method<T>(T[] input, T[] output)
void Method<T>(T[] input, int inputStartIndex, int inputLength,
      T[] output, int outputStartIndex, int outputLength)
unsafe void Method<T>(void* input, int inputLength, void* output)
unsafe void Method<T>(void* input, int startIndex, int length,
      void* output, int outputStartIndex, int outputLength)
unsafe void Method<T>(void* input, int inputLength, T[] output)
unsafe void Method<T>(void* input, int startIndex, int length,
      T[] output, int outputStartIndex, int outputLength)
// it's not a full list!
```

Simplicity!

```
void Method<T>(Span<T> input)
void Method<T>(Span<T> input, Span<T> output)
```

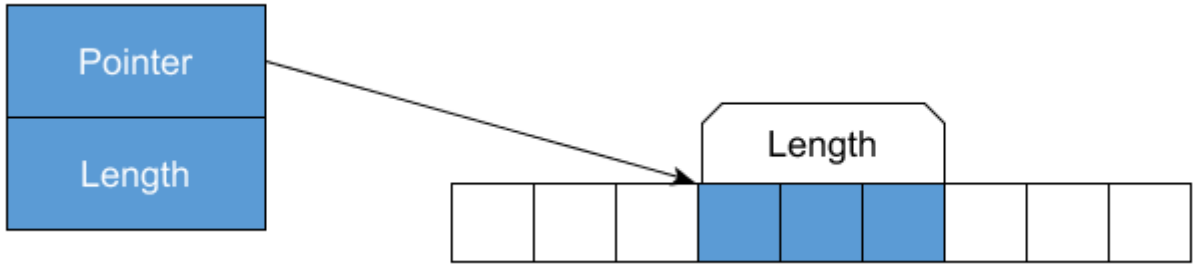
Span for existing runtimes

.NET Standard 1.0 (.NET 4.5+)



Span for new runtimes

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



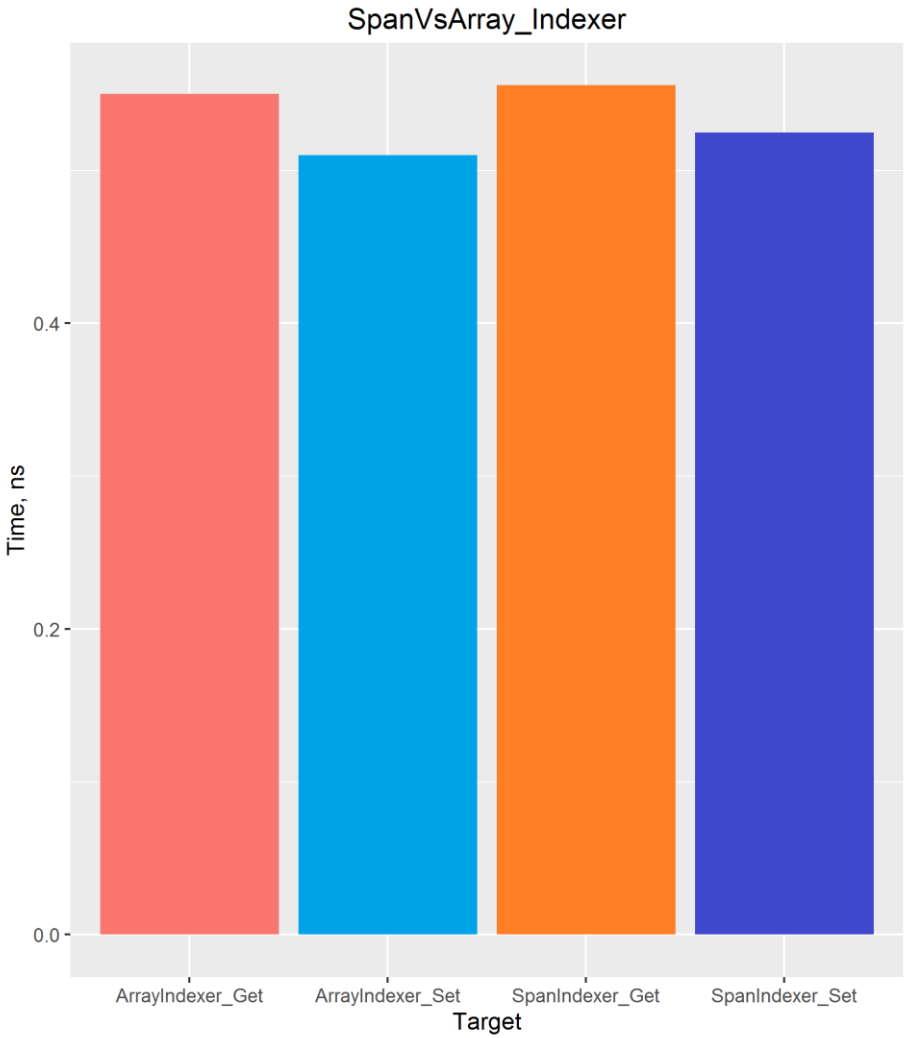
“Fast” vs “Slow” Span

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 **without allocation**!

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

Method	Mean	StdDev	Scaled	Gen 0	Allocated
Substring	10.113 ns	0.0735 ns	2.81	0.0229	48 B
Slice	3.593 ns	0.0090 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!!

Struct Tearing

- **Atomic** updates have size limits
- Some processors can't update anything > pointer atomically

```
internal class Buffer
{
    Span<byte> memory = new byte[100];

    public void Resize(int newSize)
    {
        memory = new byte[newSize];
    }

    public byte this[int index]
       => memory[index];
}
```

Struct Tearing

- **Atomic** updates have size limits
- Some processors can't update anything > pointer atomically

```
internal class Buffer
{
    Span<byte> memory = new byte[100];

    public void Resize(int newSize)
    {
        memory = new byte[newSize];
    }

    public byte this[int index]
       => memory[index];
}
```

Thread 1	memory	Thread 2

Struct Tearing

- **Atomic** updates have size limits
- Some processors can't update anything > pointer atomically

```
internal class Buffer
{
    Span<byte> memory = new byte[100];

    public void Resize(int newSize)
    {
        memory = new byte[newSize];
    }

    public byte this[int index]
       => memory[index];
}
```

Thread 1	memory	Thread 2
memory.Pointer = A;	(A, 0)	

Struct Tearing

- **Atomic** updates have size limits
- Some processors can't update anything > pointer atomically

```
internal class Buffer
{
    Span<byte> memory = new byte[100];

    public void Resize(int newSize)
    {
        memory = new byte[newSize];
    }

    public byte this[int index]
       => memory[index];
}
```

Thread 1	memory	Thread 2
memory.Pointer = A;	(A, 0)	
memory.Length = 100;	(A, 100)	

Struct Tearing

- **Atomic** updates have size limits
- Some processors can't update anything > pointer atomically

```
internal class Buffer
{
    Span<byte> memory = new byte[100];

    public void Resize(int newSize)
    {
        memory = new byte[newSize];
    }

    public byte this[int index]
       => memory[index];
}
```

Thread 1	memory	Thread 2
memory.Pointer = A;	(A, 0)	
memory.Length = 100;	(A, 100)	
memory.Pointer = B;	(B, 100)	

Struct Tearing

- **Atomic** updates have size limits
- Some processors can't update anything > pointer atomically

```
internal class Buffer
{
    Span<byte> memory = new byte[100];

    public void Resize(int newSize)
    {
        memory = new byte[newSize];
    }

    public byte this[int index]
       => memory[index];
}
```

Thread 1	memory	Thread 2
memory.Pointer = A;	(A, 0)	
memory.Length = 100;	(A, 100)	
memory.Pointer = B;	(B, 100)	read(Buffer[10]);

Struct Tearing

- **Atomic** updates have size limits
- Some processors can't update anything > pointer atomically

```
internal class Buffer
{
    Span<byte> memory = new byte[100];

    public void Resize(int newSize)
    {
        memory = new byte[newSize];
    }

    public byte this[int index]
       => memory[index];
}
```

Thread 1	memory	Thread 2
memory.Pointer = A;	(A, 0)	
memory.Length = 100;	(A, 100)	
memory.Pointer = B;	(B, 100)	read(Buffer[10]);
memory.Length = 1;	(B, 1)	

Why not to synchronize the acces?

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: ldloca.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;
```

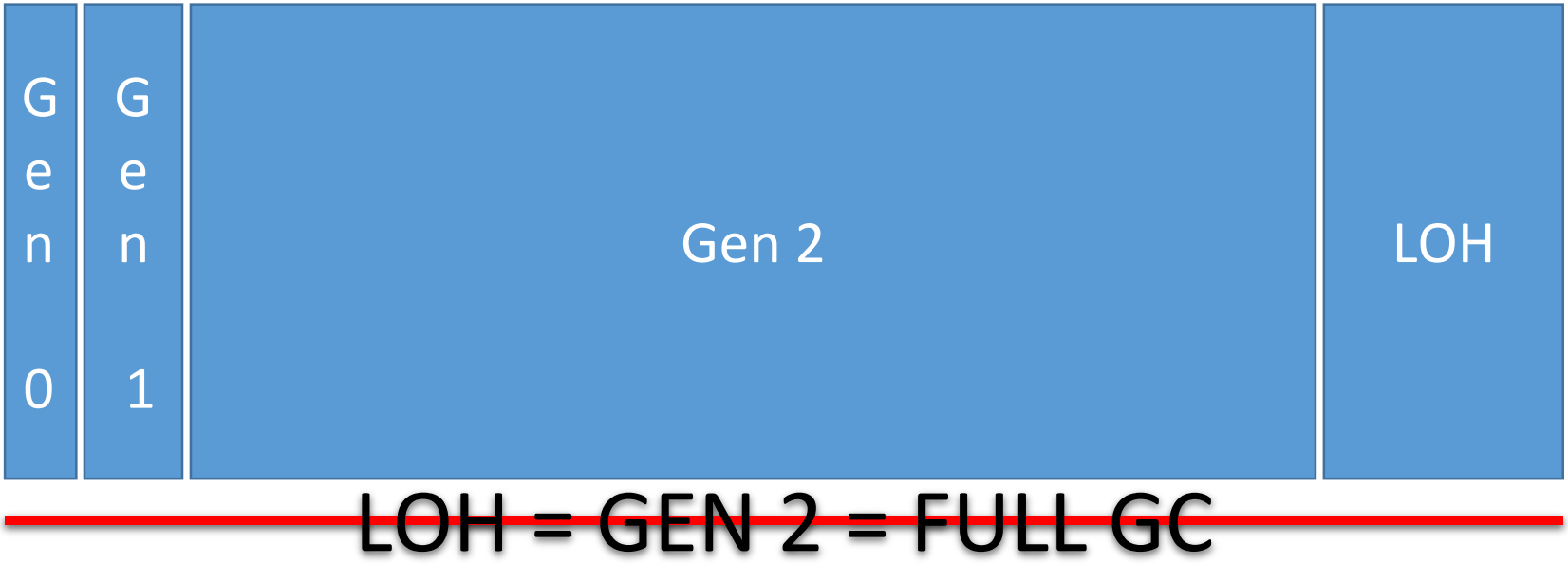
Possible solution: Pseudocode

```
abstract class SpanFactory<T> : IDisposable {
     abstract Span<T> AsSpan(int startIndex, int length);
     abstract void Dispose();
}
unsafe class MarshalSpanFactory<T> : SpanFactory<T> {
    IntPtr _pointer;
    override Span<T> AsSpan(int startIndex, int length) => new Span<T>(_pointer, startInfex, length);
    override void Dispose() => Marshal.FreeHGlobal(_pointer);
}
class ManagedPooledSpanFactory<T> : SpanFactory<T> {
    T[] _array;
    override Span<T> AsSpan(int startIndex, int length) => new Span<T>(_array, startIndex, length);
    override void Dispose() => Pool.Shared.Return(_array);
}
```

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.

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

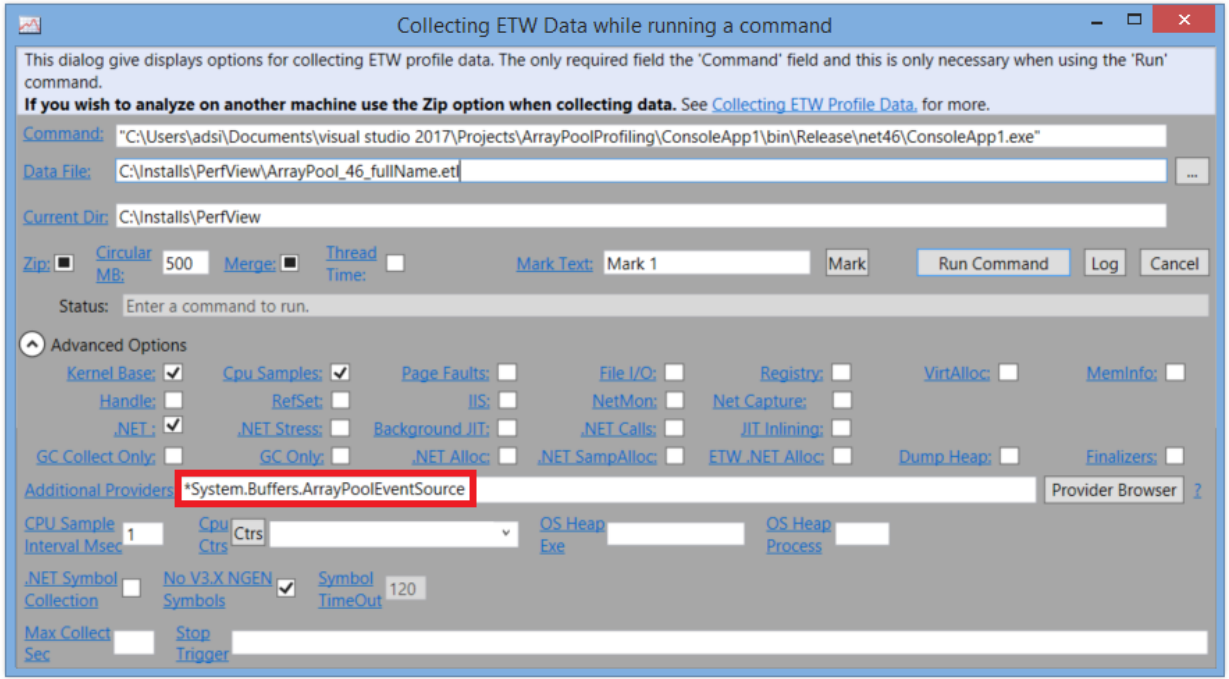
Allocate? Pool?

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**

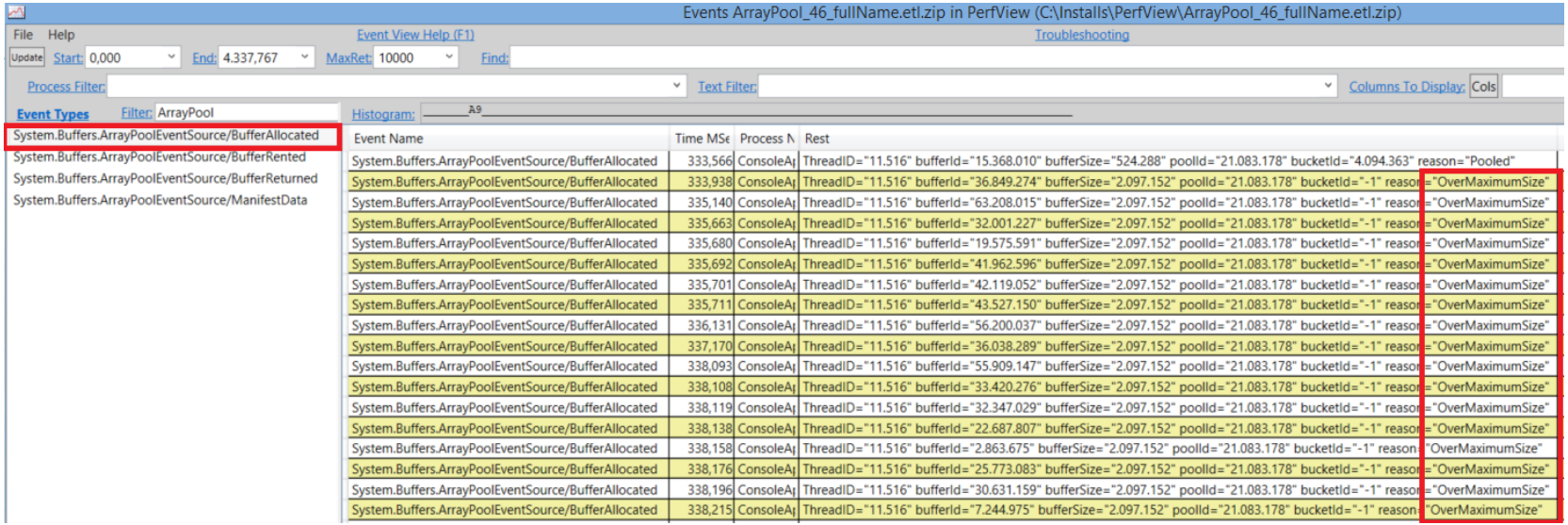
Allocate? Pool?

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!**

Async on hotpath

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

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 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	StdDev	Scaled	Gen 0	Gen 1	Allocated
ConsumeTask	100	720.9 ns	10.652 ns	1.49	3.4674	0.0001	7272 B
ConsumeValueTaskWrong	100	1,097.4 ns	15.474 ns	2.27	-	-	0 B
ConsumeValueTaskProperly	100	482.9 ns	2.758 ns	1.00	-	-	0 B
ConsumeValueTaskCrazy	100	417.0 ns	4.856 ns	0.86	-	-	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

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

C# 7.2 candidate

*"We sort of want **push unsafe code out of C#** in a way. Or more to the corner. Having more low level code that is safe and efficient. More interact with native, memory and other things like that. So 7.2 is a language release that has some features that help with that"*

- Mads Torgersen, Microsoft Build 2017

"You can go to the GitHub and see what is rumbling there."

speculate

/ˈspɛkjʊleɪt/

verb

verb: **speculate**; 3rd person present: **speculates**; past tense: **speculated**; past participle: **speculated**; gerund or present participle: **speculating**

1. form a theory or conjecture about a subject without firm evidence.
   "my colleagues speculate about my private life"
   synonimy: conjecture, theorize, form theories, hypothesize, make suppositions, postulate, guess, make guesses, surmise;
   Więcej

2. invest in stocks, property, or other ventures in the hope of gain but with the risk of loss.
   "he didn't look as though he had the money to **speculate in** shares"
   synonimy: gamble, take a risk/chance, venture, take a venture, wager; invest, play the market; *informal* have a flutter, punt
   "investors can make profits from speculating on the stock market"

Generalized `ref-like` types in source code?

```
[IsRefLike] // generated by compiler
[Obsolete("New compilers will ignore it")] // generated by compiler
ref struct TwoSpans<T>
{
    // can have ref-like instance fields
    public Span<T> first;
    public Span<T> second;
}
```

Stack-referring spans in safe code???

```
unsafe
{
    int* pointerToStack = stackalloc int[256];
    Span<int> stackMemory = new Span<int>(pointerToStack, 256+1);
}

unsafe
{
     // This usage of stackalloc does not require unsafe
     Span<byte> stackMemory = stackalloc byte[256];
}
```

C# 7.2: readonly references?

- Readonly ref parameters (aka in parameters)
- Passing readonly fields as in parameters
- Readonly ref returns
- Readonly ref locals are not planned to be supported

```
readonly ref DateTime Method(readonly ref DateTime dateTime)
{
    return ref dateTime;
}
```

C# 7.2: readonly structs?

„In short - a feature that makes all members of a struct, except constructors to have this parameter as a ref readonly.”

```
var result = readOnlyStructField.Method();
 is converted to:
var copy = readOnlyStruct;
var result = copy.Method();
```

With `readonly struct ReadOnlyStruct { /* */ }`

There would be no need for extra temp copy!

C# 7.2: ref extension methods on structs?

```
public static class Extensions
{
    public static void ExtensionMethod(
        this ref ReadOnlySpan<char> text)
    {
    }
}
```

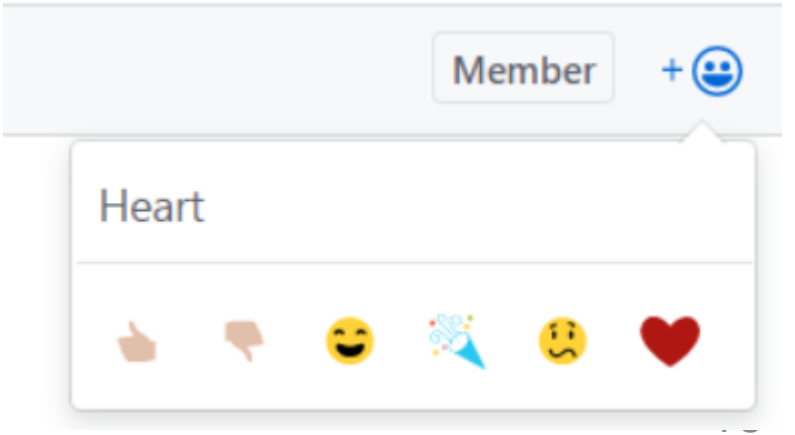
C# 7.2: blittable types??

„a type which is not a reference type and doesn't contain reference type fields at any level of nesting”

```
blitabble struct Bid
{
    public int Quantity;
    public float Price;
}

Span<TTo> Cast<TFrom, TTo>(this ref Span<TFrom> source)
    where TTo: blittable struct
    where TFrom : blittable struct

Span<byte> stream = request.Bytes;
Span<Bid> deserialized = stream.Cast<byte, Bid>();
```

Summary

- Start using Value Types today!
- Use references to avoid copying of Value Types.
- Forget about unsafe, use “ref returns and locals” instead
- Use Span and slicing to avoid allocations
- Use Span to take advantage of the native memory
- Pool the memory with ArrayPool
- Use ValueTask only if it can help you!
- Use the “Unsafe” api to use C# only
- Support C# 7.2 ideas!



Sources

- Series of Great Blog Posts by Vladimir Sadov
- Span<T> design document
- Compile time enforcement of safety for ref-like types
- ValueTask doesn't inline well- GitHub issue
- C# 7.2 Milestone on GitHub
- Readonly references proposal
- Initial blittable proposal
- Memory Systems (University Of Mary Washington)
- Latency Numbers Every Programmer Should Know (Berkeley)

Thank you!

Slides: http://adamsitnik.com/files/NDC2017.pdf

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

@SitnikAdam
Adam.Sitnik@gmail.com