Unlocking .NET Performance

Stephen Toub

Partner Software Engineer

Microsoft

.NET Performance Renaissance

- Performance is a key focus at all levels of the stack
  - Every change is viewed through a perf lense
  - Huge investments focused entirely on performance
  - The .NET community gets to vote with their feet
    - Lots of investment in reviewing PRs

- Read all about it
  - Performance Improvements in .NET Core 2.0
  - Performance Improvements in .NET Core 2.1
  - Performance Improvements in .NET Core 3.0
  - Performance Improvements in .NET 5
  - Performance Improvements in .NET 6 (haven't started writing this one yet ☺)

- Impact
  - Existing code just gets faster
  - New code benefits from employing new APIs and new patterns
    - Most new APIs also used internally

Distribution Vehicle impacting Perf

- .NET Framework is distributed with Windows, updated in-place
  - Every change can break someone, even correct changes that “just” improve perf

- .NET Core is side-by-side
  - Multiple shared frameworks can be installed at the same time
  - Apps can carry their own local copy
    - (That local copy can even be trimmed for size.)

- As a result, risk tolerance is much higher
  - “Breaking changes” are still a big deal, but fair game.
    - Breaking changes in .NET 5 - .NET | Microsoft Docs
  - We move and churn a lot faster.
  - We make both large overhauls and many small tweaks.
  - We accept lots of contributions (with appropriate review).

Open Source impacting Perf

- .NET Core 3.1 => .NET 5, more than 250 PRs focused on perf
  - > 20% were from outside of Microsoft
- .NET 5 => .NET 6, already have more than that…
- Devs improve perf for things they care about
  - Sometimes it impacts everything, e.g.
    - Use xmm for stack prolog zeroing rather than rep stos by benaadams #32538
  - Sometimes it's more niche, e.g.
    - Speed up System.Drawing.Color factory and HSB/HSL methods by saucecontrol #31838
  - Sometimes it's experimental, e.g.
    - Add large pages support in GC by mjsabby #23251
  - We ❤️ it all.

Microbenchmarks

Changes Driven By Data

Benchmark.NET

We provide / ask for microbenchmarks on *all* relevant PRs.

```
dotnet add package benchmarkdotnet
```

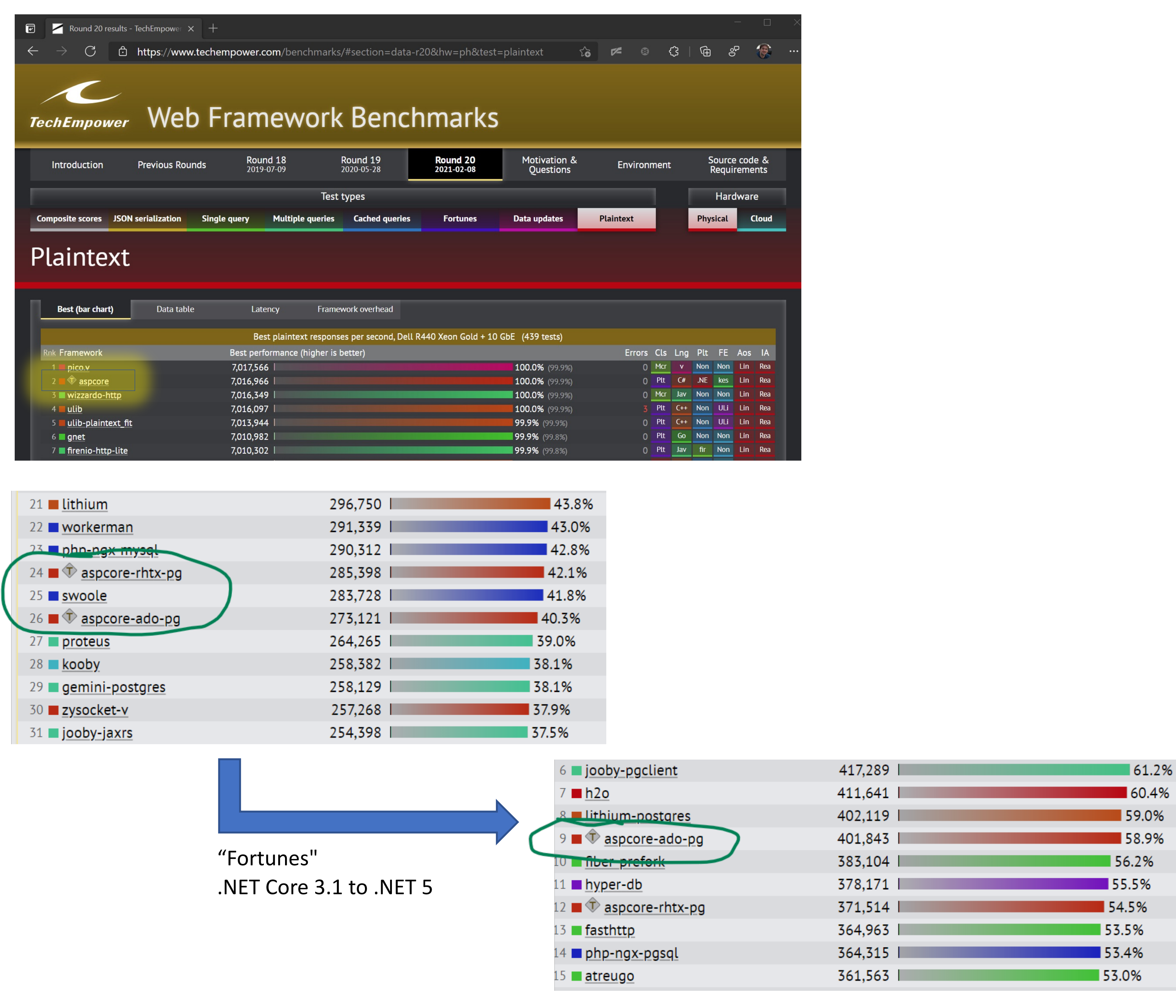
```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

[MemoryDiagnoser]
public class Program
{
    static void Main(string[] args) => BenchmarkSwitcher.FromAssembly(typeof(Program).Assembly).Run(args);

    private int _value = 12345;

    [Benchmark]
    public string Int32ToString() => _value.ToString();
}
```

```
dotnet run -c Release -f net48 --filter ** --runtimes net48 netcoreapp2.1 netcoreapp3.1 net5.0
```

Method	Runtime	Mean	Ratio	Allocated
Int32ToString	.NET Framework 4.8	48.30 ns	1.00	40 B
Int32ToString	.NET Core 2.1	22.18 ns	0.46	40 B
Int32ToString	.NET Core 3.1	21.31 ns	0.44	32 B
Int32ToString	.NET 5.0	11.79 ns	0.24	32 B

Industry Benchmarks

Changes Driven By Data



$$$

Changes Driven By Data

The “1000s of Cores” Club…

```
private Regex _email = new Regex(
    @"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|" +
     "(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,12}|[0-9]{1,3})(\]?)$", RegexOptions.Compiled);

[Benchmark]
[Arguments("someone@example.org")]
public bool IsMatch(string address) => _email.IsMatch(address);
```

Method	Runtime	Mean	Ratio
IsMatch	.NET Framework 4.8	469.6 ns	1.00
IsMatch	.NET Core 3.1	384.6 ns	0.82
IsMatch	.NET 5.0	148.6 ns	0.32

$$$

Changes Driven By Data

The “1000s of Cores” Club…

```
private Stream _compressedStream;

[GlobalSetup]
public void Setup()
{
    byte[] text = new HttpClient().GetByteArrayAsync("http://www.gutenberg.org/cache/epub/3200/pg3200.txt").Result;
    _compressedStream = new MemoryStream();
    using (var ds = new DeflateStream(_compressedStream, CompressionLevel.Optimal, leaveOpen: true))
    {
        ds.Write(text, 0, text.Length);
    }
}

[Benchmark]
public async Task Decompress()
{
    _compressedStream.Position = 0;
    using (var ds = new DeflateStream(_compressedStream, CompressionMode.Decompress, leaveOpen: true))
    {
        await ds.CopyToAsync(Stream.Null);
    }
}
```

Method	Runtime	Ratio	Allocated
Decompress	.NET Framework 4.8	1.00	353,500 B
Decompress	.NET 5.0	0.82	302 B

Includes a 3x gain ported back to .NET Framework 4.8

$$$

Changes Driven By Data

The “1000s of Cores” Club…

```
private string[] _keys;
private Dictionary<string, int> _dictionary = new Dictionary<string, int>();

[GlobalSetup]
public void Setup()
{
    _keys = Enumerable.Range(0, 1000).Select(i => Path.GetRandomFileName()).ToArray();
    _dictionary = _keys.Select((key, i) => (key, i))
                       .ToDictionary(t => t.key, t => t.i, StringComparer.OrdinalIgnoreCase);
}

[Benchmark]
public int TryGetValue()
{
    int sum = 0;
    foreach (string key in _keys)
        if (_dictionary.TryGetValue(key, out int value))
            sum += value;
    return sum;
}
```

Method	Runtime	Mean	Ratio
TryGetValue	.NET Framework 4.8	57.89 us	1.00
TryGetValue	.NET Core 3.1	31.29 us	0.54
TryGetValue	.NET 5.0	19.56 us	0.34

$$$

Changes Driven By Data

The “1000s of Cores” Club…

```
[Benchmark]
public string Roundtrip() =>
    Encoding.UTF8.GetString(
        Encoding.UTF8.GetBytes(
            "Free. Cross-platform. Open source. A developer platform for building all your apps."));
```

Method	Runtime	Mean	Ratio	Allocated
Roundtrip	.NET Framework 4.8	190.02 ns	1.00	304 B
Roundtrip	.NET 5.0	79.90 ns	0.42	304 B

Many different ways these improvements are achieved…

Sometimes via complete rewrites.

e.g. ConcurrentQueue<T>

Rewritten in .NET Core 2.1

Focus on reducing synchronization and enabling memory reuse

Now the ThreadPool's global queue

```
private ConcurrentQueue<int> _queue = new ConcurrentQueue<int>();
private Barrier _barrier = new Barrier(2);

[Benchmark]
public async Task PC()
{
    await Task.WhenAll(
        Task.Run(() =>
        {
            _barrier.SignalAndWait();
            for (int i = 0; i < 1_000_000; i++)
                _queue.Enqueue(i);
        }),
        Task.Run(() =>
        {
            _barrier.SignalAndWait();
            for (int i = 0; i < 1_000_000; i++)
            {
                while (!_queue.TryDequeue(out _)) ;
            }
        }));
}
```

Method	Runtime	Mean	Ratio	Allocated
PC	.NET Framework 4.8	44.99 ms	1.00	8,534,941 B
PC	.NET 5.0	11.80 ms	0.26	396 B

Sometimes via changing algorithmic complexity.

e.g. LINQ, passing info between operators

```
private int[] _data;

[GlobalSetup]
public void Setup()
{
    var r = new Random();
    _data = new int[10_000_000];
    for (int i = 0; i < _data.Length; i++) _data[i] = r.Next();
}

[Benchmark]
public int LINQ() => _data.OrderBy(i => i).Skip(10).Take(1).Sum();
```

Method	Runtime	Mean	Ratio	Allocated
LINQ	.NET Framework 4.8	3,742.1 ms	1.00	114 MB
LINQ	.NET 5.0	211.0 ms	0.06	114 MB

Sometimes via pervasive changes to how we write code.

{ReadOnly}Span<T>

Coding Pattern Changes

```
public readonly ref struct Span<T>
{
    private readonly ref T _pointer;
    private readonly int _length;
}
```

Zero-alloc representation of contiguous memory

- `ReadOnlySpan<char> s = string;`
- `Span<Person> s = array;`
- `Span<byte> s = stackalloc byte[123];`
- `Span<int> s = new Span<int>((int*)ptr, length);`

Indexing, e.g. `Person p = span[i];`

Slicing, e.g. `span = span[1..^1]; // span.Slice(1, span.Length - 2)`

Reinterpreting, `e.g. MemoryMarshal.AsBytes(spanOfInt32)`

Tons of methods that operate on spans

Performance + Safety

Roslyn + JIT both optimize for spans

More and more optimizations added in every release, based on usage patterns

{ReadOnly}Span<T>

Coding Pattern Changes

```
[Benchmark(Baseline = true)]
[Arguments("\"Stephen\"")]
public string SayHello1(string quotedName)
{
    return string.Concat("Hello, ", quotedName.Substring(1, quoteName.Length – 2));
}
```

Method	Runtime	Mean	Allocated
SayHello1	.NET Framework 4.8	25.34 ns	96 B
SayHello1	.NET 5.0	22.51 ns	96 B

```
[Benchmark]
[Arguments("\"Stephen\"")]
[SkipLocalsInit]
public string SayHello2(string quotedName)
{
    Span<char> span = stackalloc char[256];

    "Hello, ".AsSpan().CopyTo(span);
    quotedName.AsSpan()[1..^1].CopyTo(span.Slice("Hello, ".Length));

    return span.Slice(0, "Hello, ".Length + quotedName.Length - 2).ToString();
}
```

Method	Mean	Allocated
SayHello2	18.22 ns	56 B

```
[Benchmark]
[Arguments("\"Stephen\"")]
public string SayHello3(string quotedName)
{
    return string.Concat("Hello, ", quotedName.AsSpan()[1..^1]);
}
```

Method	Mean	Allocated
SayHello3	12.61 ns	56 B

C/C++ => C#

Coding Pattern Changes

```
public static void Sort<T>(this System.Span<T> span, System.Comparison<T> comparison) { }
public static void Sort<TKey, TValue>(this System.Span<TKey> keys, System.Span<TValue> items) { }
public static void Sort<TKey, TValue>(this System.Span<TKey> keys, System.Span<TValue> items, System.Comparison<TKe
public static void Sort<T, TComparer>(this System.Span<T> span, TComparer comparer) where TComparer : System.Collec
public static void Sort<TKey, TValue, TComparer>(this System.Span<TKey> keys, System.Span<TValue> items, TComparer
```

```
private int[] _orig = Enumerable.Range(0, 10).Reverse().ToArray();
private int[] _array = new int[10];

[Benchmark]
public void Sort()
{
    _orig.CopyTo(_array, 0);
    Array.Sort(_array);
}
```

Method	Runtime	Mean	Ratio
Sort	.NET Framework 4.8	95.07 ns	1.00
Sort	.NET Core 3.1	85.78 ns	0.90
Sort	.NET 5.0	53.98 ns	0.57

C/C++ => C#

Coding Pattern Changes

GC pause time

```
using System;
using System.Diagnostics;
using System.Threading;

class Program
{
    public static void Main()
    {
        new Thread(() =>
        {
            var a = new int[20];
            while (true) Array.Sort(a);
        }) { IsBackground = true }.Start();

        var sw = new Stopwatch();
        while (true)
        {
            sw.Restart();
            for (int i = 0; i < 10; i++)
            {
                GC.Collect();
                Thread.Sleep(15);
            }
            Console.WriteLine(sw.Elapsed.TotalSeconds);
        }
    }
}
```

```
C:\users\stoub\Desktop\Benchmarks> dotnet run -c Release -f netcoreapp3.1
2.7673352
3.3650662
4.9482748
4.933966
4.4339374
```

```
C:\users\stoub\Desktop\Benchmarks> dotnet run -c Release -f net5.0
0.2045423
0.2497763
0.2034448
0.2484892
0.189885
0.2338674
```

ValueTask

Coding Pattern Changes

Runtime async support completely rewritten in .NET Core 2.1…

```
private AsyncLocal<int> _asyncLocal = new AsyncLocal<int>();

[Benchmark]
public async Task YieldMany()
{
    _asyncLocal.Value = 42;
    for (int i = 0; i < 1000; i++)
        await YieldOnce();
}

private static async Task YieldOnce() =>
    await Task.Yield();
```

YieldMany	.NET Framework 4.8	1,924.3 us	1.00	634,497 B
YieldMany	.NET Core 2.1	928.1 us	0.48	112,280 B
YieldMany	.NET Core 3.1	822.0 us	0.43	112,280 B
YieldMany	.NET 5.0	814.5 us	0.42	112,280 B

.NET Framework 4.8

Type	Allocations	Bytes	Average Size (Bytes)
System.Threading.ExecutionContext	2,001	176,088	88
System.Action	2,000	128,000	64
System.Runtime.CompilerServices.AsyncMethodBuilderCore.MoveNextRunner	2,000	64,000	32
System.Threading.Tasks.Task<System.Threading.Tasks.VoidTaskResult>	1,002	80,160	80
System.Runtime.Remoting.Messaging.LogicalCallContext	1,000	72,000	72
Program.<YieldOnce>d__2	1,000	56,000	56
System.Threading.QueueUserWorkItemCallback	1,000	40,000	40
System.Threading.Tasks.AwaitTaskContinuation	993	39,720	40

.NET 5.0

Type	Allocations	Bytes	Average Size (Bytes)
System.Runtime.CompilerServices.AsyncTaskMethodBuilder<System.Threading.Tasks.VoidTaskResult>.AsyncStateMachineBox<<YieldOnce>d__2>	1,000	112,000	112

…but still allocates a Task per async operation…

ValueTask

Coding Pattern Changes

ValueTask<T>

Zero alloc for **synchronously** completing operations

Possible to avoid allocation even for **asynchronously** completing operations

IValueTaskSource{<T>}

```
private Socket _server, _client;
private Memory<byte> _buffer = new Memory<byte>(new byte[1]);

[GlobalSetup]
public void Setup()
{
    using (var listener = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp))
    {
        listener.Bind(new IPEndPoint(IPAddress.Loopback, 0));
        listener.Listen(1);

        _client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        _client.Connect(listener.LocalEndPoint);

        _server = listener.Accept();
    }
}

[Benchmark]
public async Task ReceiveSend()
{
    for (int i = 0; i < 1000; i++)
    {
        var recv = _server.ReceiveAsync(_buffer, SocketFlags.None);
        await _client.SendAsync(_buffer, SocketFlags.None);
        await recv;
    }
}
```

Method	Runtime	Mean	Ratio	Allocated
ReceiveSend	.NET Framework 4.8	35.43 ms	1.00	333 KB
ReceiveSend	.NET Core 2.1	21.08 ms	0.59	-
ReceiveSend	.NET Core 3.1	20.09 ms	0.57	-
ReceiveSend	.NET 5.0	18.64 ms	0.53	-

ArrayPool

Coding Pattern Changes

ArrayPool<T>.Shared.Rent/Return

Used throughout core libraries

```
static async Task CopyToAsync(Stream source, Stream destination, int bufferSize, CancellationToken cancellationToken)
{
    byte[] buffer = ArrayPool<byte>.Shared.Rent(bufferSize);
    try
    {
        int bytesRead;
        while ((bytesRead = await source.ReadAsync(buffer, cancellationToken)) != 0)
        {
            await destination.WriteAsync(new ReadOnlyMemory<byte>(buffer, 0, bytesRead), cancellationToken);
        }
    }
    finally
    {
        ArrayPool<byte>.Shared.Return(buffer);
    }
}
```

Hardware Intrinsics

Coding Pattern Changes

Vector, Vector<T>, Vector64<T>, Vector128<T>, Vector256<T>, …

System.Runtime.Intrinsics

- Arm: AdvSimd, Aes, Crc32, Dp, Rdm, Sha1, Sha256
- X86: Aes, Avx, Avx2, Bmi1, Bmi2, Fma, Lzcnt, Pclmulqdq, Popcnt, Sse, Sse2, Sse3, Sse41, Sse42, Ssse3

Used in

- Span, String, BitArray, Base64, Encoding.UTF8/UTF16/ASCII/Latin1, JsonSerializer, WebSocket, …

```
private string _text =
    "Shall I compare thee to a summer's day? " +
    "Thou art more lovely and more temperate: " +
    "Rough winds do shake the darling buds of May, " +
    "Sometime too hot the eye of heaven shines,";

[Benchmark]
public bool Contains() => _text.Contains("shines", StringComparison.Ordinal);
```

Method	Runtime	Mean	Ratio
IndexOf	.NET Framework 4.8	41.283 ns	1.00
IndexOf	.NET Core 2.1	9.942 ns	0.24
IndexOf	.NET Core 3.1	7.009 ns	0.17
IndexOf	.NET 5.0	6.004 ns	0.15

And much more…

Thanks!

- Still on .NET Framework?
  - Start thinking about moving to .NET Core.
- Already on .NET Core?
  - Nice! Stay current.

- Get involved!
  - http://github.com/dotnet/runtime
  - Things you want to see improved? Submit issues. Better yet, submit PRs!

Q&A