Performance Improvements in .NET 7

Stephen Toub

Partner Software Engineer, .NET

Microsoft

Introduction

A year ago, I published [Performance Improvements in .NET 6](), following on the heels of similar posts for [.NET 5](), [.NET Core 3.0](), [.NET Core 2.1](), and [.NET Core 2.0](). I enjoy writing these posts and love reading developers' responses to them. One comment in particular last year resonated with me. The commenter cited the Die Hard movie quote, "'When Alexander saw the breadth of his domain, he wept for there were no more worlds to conquer'," and questioned whether .NET performance improvements were similar. Has the well run dry? Are there no more "[performance] worlds to conquer"? I'm a bit giddy to say that, even with how fast .NET 6 is, .NET 7 definitively highlights how much more can be and has been done.

As with previous versions of .NET, performance is a key focus that pervades the entire stack, whether it be features created explicitly for performance or non-performance-related features that are still designed and implemented with performance keenly in mind. And now that a .NET 7 release candidate is just around the corner, it's a good time to discuss much of it. Over the course of the last year, every time I've reviewed a PR that might positively impact performance, I've copied that link to a journal I maintain for the purposes of writing this post. When I sat down to write this a few weeks ago, I was faced with a list of almost 1000 performance-impacting PRs (out of more than 7000 PRs that went into the release), and I'm excited to share approximately 500 of them here with you.

One thought before we dive in. In past years, I've received the odd piece of negative feedback about the length of some of my performance-focused write-ups, and while I disagree with the criticism, I respect the opinion. So, this year, consider this a "choose your own adventure." If you're here just looking for a super short adventure, one that provides the top-level summary and a core message to take away from your time here, I'm happy to oblige:

TL;DR: .NET 7 is fast. Really fast. A thousand performance-impacting PRs went into runtime and core libraries this release, never mind all the improvements in ASP.NET Core and Windows Forms and Entity Framework and beyond. It's the fastest .NET ever. If your manager asks you why your project should upgrade to .NET 7, you can say "in addition to all the new functionality in the release, .NET 7 is super fast."

Or, if you prefer a slightly longer adventure, one filled with interesting nuggets of performance-focused data, consider skimming through the post, looking for the small code snippets and corresponding tables showing a wealth of measurable performance improvements. At that point, you, too, may walk away with your head held high and my thanks.

Both noted paths achieve one of my primary goals for spending the time to write these posts, to highlight the greatness of the next release and to encourage everyone to give it a try. But, I have other goals for these posts, too. I want everyone interested to walk away from this post with an upleveled understanding of how .NET is implemented, why various decisions were made, tradeoffs that were evaluated, techniques that were employed, algorithms that were considered, and valuable tools and approaches that were utilized to make .NET even faster than it was previously. I want developers to learn from our own learnings and find ways to apply this new-found knowledge to their own codebases, thereby further increasing the overall performance of code in the ecosystem. I want developers to take an extra beat, think about reaching for a profiler the next time they're working on a gnarly problem, think about looking at the source for the component they're using in order to better understand how to work with it, and think about revisiting previous assumptions and decisions to determine whether they're still accurate and appropriate. And I want developers to be excited at the prospect of submitting PRs to improve .NET not only for themselves but for every developer around the globe using .NET. If any of that sounds interesting, then I encourage you to choose the last adventure: prepare a carafe of your favorite hot beverage, get comfortable, and please enjoy.

Contents

Setup

The microbenchmarks throughout this post utilize benchmarkdotnet. To make it easy for you to follow along with your own validation, I have a very simple setup for the benchmarks I use. Create a new C# project:

```
dotnet new console -o benchmarks
cd benchmarks
```

Your new `benchmarks` directory will contain a `benchmarks.csproj` file and a `Program.cs` file. Replace the contents of `benchmarks.csproj` with this:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>net7.0;net6.0</TargetFrameworks>
    <LangVersion>Preview</LangVersion>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
    <ServerGarbageCollection>true</ServerGarbageCollection>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="benchmarkdotnet" Version="0.13.2" />
  </ItemGroup>

</Project>
```

and the contents of `Program.cs` with this:

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using Microsoft.Win32;
using System;
using System.Buffers;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.IO.Compression;
using System.IO.MemoryMappedFiles;
using System.IO.Pipes;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Net.Security;
using System.Net.Sockets;
using System.Numerics;
```

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Intrinsics;
using System.Security.Authentication;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;
using System.Text;
using System.Text.Json;
using System.Text.RegularExpressions;
using System.Threading;
using System.Threading.Tasks;
using System.Xml;

[MemoryDiagnoser(displayGenColumns: false)]
[DisassemblyDiagnoser]
[HideColumns("Error", "StdDev", "Median", "RatioSD")]
public partial class Program
{
    static void Main(string[] args) =>
BenchmarkSwitcher.FromAssembly(typeof(Program).Assembly).Run(args);

    // ... copy [Benchmark]s here
}
```

For each benchmark included in this write-up, you can then just copy and paste the code into this test class, and run the benchmarks. For example, to run a benchmark comparing performance on .NET 6 and .NET 7, do:

```
dotnet run -c Release -f net6.0 --filter '**' --runtimes net6.0 net7.0
```

This command says "build the benchmarks in release configuration targeting the .NET 6 surface area, and then run all of the benchmarks on both .NET 6 and .NET 7." Or to run just on .NET 7:

```
dotnet run -c Release -f net7.0 --filter '**' --runtimes net7.0
```

which instead builds targeting the .NET 7 surface area and then only runs once against .NET 7. You can do this on any of Windows, Linux, or macOS. Unless otherwise called out (e.g. where the improvements are specific to Unix and I run the benchmarks on Linux), the results I share were recorded on Windows 11 64-bit but aren't Windows-specific and should show similar relative differences on the other operating systems as well.

The release of the first .NET 7 release candidate is right around the corner. All of the measurements in this post were gathered with a recent daily build of .NET 7 RC1.

Also, my standard caveat: These are microbenchmarks. It is expected that different hardware, different versions of operating systems, and the way in which the wind is currently blowing can affect the numbers involved. Your mileage may vary.

JIT

I'd like to kick off a discussion of performance improvements in the Just-In-Time (JIT) compiler by talking about something that itself isn't actually a performance improvement. Being able to understand exactly what assembly code is generated by the JIT is critical when fine-tuning lower-level, performance-sensitive code. There are multiple ways to get at that assembly code. The online tool sharplab.io is *incredibly useful* for this (thanks to [@ashmind](https://github.com/ashmind) for this tool); however it currently only targets a single release, so as I write this I'm only able to see the output for .NET 6, which makes it difficult to use for A/B comparisons. godbolt.org is also valuable for this, with C# support added in compiler-explorer/compiler-explorer#3168 from [@hez2010](https://github.com/hez2010), with similar limitations. The most flexible solutions involve getting at that assembly code locally, as it enables comparing whatever versions or local builds you desire with whatever configurations and switches set that you need.

One common approach is to use the `[DisassemblyDiagnoser]` in benchmarkdotnet. Simply slap the `[DisassemblyDiagnoser]` attribute onto your test class: benchmarkdotnet will find the assembly code generated for your tests and some depth of functions they call, and dump out the found assembly code in a human-readable form. For example, if I run this test:

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

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

    private int _a = 42, _b = 84;

    [Benchmark]
    public int Min() => Math.Min(_a, _b);
}
```

with:

```
dotnet run -c Release -f net7.0 --filter '**'
```

in addition to doing all of its normal test execution and timing, benchmarkdotnet also outputs a `Program-asm.md` file that contains this:

```
; Program.Min()
       mov       eax,[rcx+8]
       mov       edx,[rcx+0C]
       cmp       eax,edx
```

```
        jg        short M00_L01
        mov       edx,eax
M00_L00:
        mov       eax,edx
        ret
M00_L01:
        jmp       short M00_L00
; Total bytes of code 17
```

Pretty neat. This support was recently improved further in [dotnet/benchmarkdotnet#2072](), which allows passing a filter list on the command-line to benchmarkdotnet to tell it exactly which methods' assembly code should be dumped.

If you can get your hands on a "debug" or "checked" build of the .NET runtime ("checked" is a build that has optimizations enabled but also still includes asserts), and specifically of clrjit.dll, another valuable approach is to set an environment variable that causes the JIT itself to spit out a human-readable description of all of the assembly code it emits. This can be used with any kind of application, as it's part of the JIT itself rather than part of any specific tool or other environment, it supports showing the code the JIT generates each time it generates code (e.g. if it first compiles a method without optimization and then later recompiles it with optimization), and overall it's the most accurate picture of the assembly code as it comes "straight from the horses mouth," as it were. The (big) downside of course is that it requires a non-release build of the runtime, which typically means you need to build it yourself from the sources in the dotnet/runtime repo.

… until .NET 7, that is. As of dotnet/runtime#73365, this assembly dumping support is now available in release builds as well, which means it's simply part of .NET 7 and you don't need anything special to use it. To see this, try creating a simple "hello world" app like:

```
using System;

class Program
{
    public static void Main() => Console.WriteLine("Hello, world!");
}
```

and building it (e.g. `dotnet build -c Release`). Then, set the `DOTNET_JitDisasm` environment variable to the name of the method we care about, in this case "Main" (the exact syntax allowed is more permissive and allows for some use of wildcards, optional namespace and class names, etc.). As I'm using PowerShell, that means:

```
$env:DOTNET_JitDisasm="Main"
```

and then running the app. You should see code like this output to the console:

```
; Assembly listing for method Program:Main()
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-0 compilation
; MinOpts code
; rbp based frame
; partially interruptible

G_M000_IG01:                ;; offset=0000H
       55                   push     rbp
       4883EC20             sub      rsp, 32
```

```
       488D6C2420           lea      rbp, [rsp+20H]

G_M000_IG02:                ;; offset=000AH
       48B9D820400A8E010000 mov      rcx, 0x18E0A4020D8
       488B09               mov      rcx, gword ptr [rcx]
       FF1583B31000         call     [Console:WriteLine(String)]
       90                   nop

G_M000_IG03:                ;; offset=001EH
       4883C420             add      rsp, 32
       5D                   pop      rbp
       C3                   ret

; Total bytes of code 36

Hello, world!
```

This is immeasurably helpful for performance analysis and tuning, even for questions as simple as "did my function get inlined" or "is this code I expected to be optimized away actually getting optimized away." Throughout the rest of this post, I'll include assembly snippets generated by one of these two mechanisms, in order to help exemplify concepts.

Note that it can sometimes be a little confusing figuring out what name to specify as the value for `DOTNET_JitDisasm`, especially when the method you care about is one that the C# compiler names or name mangles (since the JIT only sees the IL and metadata, not the original C#), e.g. the name of the entry point method for a program with top-level statements, the names of local functions, etc. To both help with this and to provide a really valuable top-level view of the work the JIT is doing, .NET 7 also supports the new `DOTNET_JitDisasmSummary` environment variable (introduced in dotnet/runtime#74090). Set that to "1", and it'll result in the JIT emitting a line every time it compiles a method, including the name of that method which is copy/pasteable with `DOTNET_JitDisasm`. This feature is useful in-and-of-itself, however, as it can quickly highlight for you what's being compiled, when, and with what settings. For example, if I set the environment variable and then run a "hello, world" console app, I get this output:

```
   1: JIT compiled CastHelpers:StelemRef(Array,long,Object) [Tier1, IL size=88, code
size=93]
   2: JIT compiled CastHelpers:LdelemaRef(Array,long,long):byref [Tier1, IL size=44, code
size=44]
   3: JIT compiled SpanHelpers:IndexOfNullCharacter(byref):int [Tier1, IL size=792, code
size=388]
   4: JIT compiled Program:Main() [Tier0, IL size=11, code size=36]
   5: JIT compiled ASCIIUtility:NarrowUtf16ToAscii(long,long,long):long [Tier0, IL
size=490, code size=1187]
Hello, world!
```

We can see for "hello, world" there's only 5 methods that actually get JIT compiled. There are of course many more methods that get executed as part of a simple "hello, world," but almost all of them have precompiled native code available as part of the ["Ready To Run" (R2R)]() images of the core libraries. The first three in the above list (`StelemRef`, `LdelemaRef`, and `IndexOfNullCharacter`) don't because they explicitly opted-out of R2R via use of the `[MethodImpl(MethodImplOptions.AggressiveOptimization)]` attribute (despite the name, this attribute should almost never be used, and is only used for very specific reasons in a few very specific places in the core libraries). Then there's our `Main` method. And lastly there's the `NarrowUtf16ToAscii` method, which doesn't have R2R code, either, due to using the variable-width `Vector<T>` (more on that later). Every other method that's run doesn't require JIT'ing. If we instead first set the `DOTNET_ReadyToRun` environment variable to `0`, the list is much longer, and gives you a very good sense of what the JIT needs to do on startup (and why technologies like R2R are important for startup time). Note how many methods get compiled before "hello, world" is output:

```
   1: JIT compiled CastHelpers:StelemRef(Array,long,Object) [Tier1, IL size=88, code
size=93]
   2: JIT compiled CastHelpers:LdelemaRef(Array,long,long):byref [Tier1, IL size=44, code
size=44]
   3: JIT compiled AppContext:Setup(long,long,int) [Tier0, IL size=68, code size=275]
   4: JIT compiled Dictionary`2:.ctor(int):this [Tier0, IL size=9, code size=40]
   5: JIT compiled Dictionary`2:.ctor(int,IEqualityComparer`1):this [Tier0, IL size=102,
code size=444]
   6: JIT compiled Object:.ctor():this [Tier0, IL size=1, code size=10]
   7: JIT compiled Dictionary`2:Initialize(int):int:this [Tier0, IL size=56, code size=231]
   8: JIT compiled HashHelpers:GetPrime(int):int [Tier0, IL size=83, code size=379]
   9: JIT compiled HashHelpers:.cctor() [Tier0, IL size=24, code size=102]
  10: JIT compiled HashHelpers:GetFastModMultiplier(int):long [Tier0, IL size=9, code
size=37]
  11: JIT compiled Type:GetTypeFromHandle(RuntimeTypeHandle):Type [Tier0, IL size=8, code
size=14]
  12: JIT compiled Type:op_Equality(Type,Type):bool [Tier0, IL size=38, code size=143]
  13: JIT compiled
NonRandomizedStringEqualityComparer:GetStringComparer(Object):IEqualityComparer`1 [Tier0,
IL size=39, code size=170]
  14: JIT compiled NonRandomizedStringEqualityComparer:.cctor() [Tier0, IL size=46, code
size=232]
  15: JIT compiled EqualityComparer`1:get_Default():EqualityComparer`1 [Tier0, IL size=6,
code size=36]
  16: JIT compiled EqualityComparer`1:.cctor() [Tier0, IL size=26, code size=125]
  17: JIT compiled ComparerHelpers:CreateDefaultEqualityComparer(Type):Object [Tier0, IL
size=235, code size=949]
  18: JIT compiled CastHelpers:ChkCastClass(long,Object):Object [Tier0, IL size=22, code
size=72]
  19: JIT compiled RuntimeHelpers:GetMethodTable(Object):long [Tier0, IL size=11, code
size=33]
  20: JIT compiled CastHelpers:IsInstanceOfClass(long,Object):Object [Tier0, IL size=97,
code size=257]
  21: JIT compiled GenericEqualityComparer`1:.ctor():this [Tier0, IL size=7, code size=31]
  22: JIT compiled EqualityComparer`1:.ctor():this [Tier0, IL size=7, code size=31]
  23: JIT compiled CastHelpers:ChkCastClassSpecial(long,Object):Object [Tier0, IL size=87,
code size=246]
  24: JIT compiled OrdinalComparer:.ctor(IEqualityComparer`1):this [Tier0, IL size=8, code
size=39]
  25: JIT compiled NonRandomizedStringEqualityComparer:.ctor(IEqualityComparer`1):this
[Tier0, IL size=14, code size=52]
  26: JIT compiled StringComparer:get_Ordinal():StringComparer [Tier0, IL size=6, code
size=49]
  27: JIT compiled OrdinalCaseSensitiveComparer:.cctor() [Tier0, IL size=11, code size=71]
  28: JIT compiled OrdinalCaseSensitiveComparer:.ctor():this [Tier0, IL size=8, code
size=33]
  29: JIT compiled OrdinalComparer:.ctor(bool):this [Tier0, IL size=14, code size=43]
  30: JIT compiled StringComparer:.ctor():this [Tier0, IL size=7, code size=31]
  31: JIT compiled StringComparer:get_OrdinalIgnoreCase():StringComparer [Tier0, IL size=6,
code size=49]
  32: JIT compiled OrdinalIgnoreCaseComparer:.cctor() [Tier0, IL size=11, code size=71]
  33: JIT compiled OrdinalIgnoreCaseComparer:.ctor():this [Tier0, IL size=8, code size=36]
  34: JIT compiled OrdinalIgnoreCaseComparer:.ctor(IEqualityComparer`1):this [Tier0, IL
```

```
size=8, code size=39]
  35: JIT compiled CastHelpers:ChkCastAny(long,Object):Object [Tier0, IL size=38, code
size=115]
  36: JIT compiled CastHelpers:TryGet(long,long):int [Tier0, IL size=129, code size=308]
  37: JIT compiled CastHelpers:TableData(ref):byref [Tier0, IL size=7, code size=31]
  38: JIT compiled MemoryMarshal:GetArrayDataReference(ref):byref [Tier0, IL size=7, code
size=24]
  39: JIT compiled CastHelpers:KeyToBucket(byref,long,long):int [Tier0, IL size=38, code
size=87]
  40: JIT compiled CastHelpers:HashShift(byref):int [Tier0, IL size=3, code size=16]
  41: JIT compiled BitOperations:RotateLeft(long,int):long [Tier0, IL size=17, code
size=23]
  42: JIT compiled CastHelpers:Element(byref,int):byref [Tier0, IL size=15, code size=33]
  43: JIT compiled Volatile:Read(byref):int [Tier0, IL size=6, code size=16]
  44: JIT compiled String:Ctor(long):String [Tier0, IL size=57, code size=155]
  45: JIT compiled String:wcslen(long):int [Tier0, IL size=7, code size=31]
  46: JIT compiled SpanHelpers:IndexOfNullCharacter(byref):int [Tier1, IL size=792, code
size=388]
  47: JIT compiled String:get_Length():int:this [Tier0, IL size=7, code size=17]
  48: JIT compiled Buffer:Memmove(byref,byref,long) [Tier0, IL size=59, code size=102]
  49: JIT compiled RuntimeHelpers:IsReferenceOrContainsReferences():bool [Tier0, IL size=2,
code size=8]
  50: JIT compiled Buffer:Memmove(byref,byref,long) [Tier0, IL size=480, code size=678]
  51: JIT compiled Dictionary`2:Add(__Canon,__Canon):this [Tier0, IL size=11, code size=55]
  52: JIT compiled Dictionary`2:TryInsert(__Canon,__Canon,ubyte):bool:this [Tier0, IL
size=675, code size=2467]
  53: JIT compiled OrdinalComparer:GetHashCode(String):int:this [Tier0, IL size=7, code
size=37]
  54: JIT compiled String:GetNonRandomizedHashCode():int:this [Tier0, IL size=110, code
size=290]
  55: JIT compiled BitOperations:RotateLeft(int,int):int [Tier0, IL size=17, code size=20]
  56: JIT compiled Dictionary`2:GetBucket(int):byref:this [Tier0, IL size=29, code size=90]
  57: JIT compiled HashHelpers:FastMod(int,int,long):int [Tier0, IL size=20, code size=70]
  58: JIT compiled Type:get_IsValueType():bool:this [Tier0, IL size=7, code size=39]
  59: JIT compiled RuntimeType:IsValueTypeImpl():bool:this [Tier0, IL size=54, code
size=158]
  60: JIT compiled RuntimeType:GetNativeTypeHandle():TypeHandle:this [Tier0, IL size=12,
code size=48]
  61: JIT compiled TypeHandle:.ctor(long):this [Tier0, IL size=8, code size=25]
  62: JIT compiled TypeHandle:get_IsTypeDesc():bool:this [Tier0, IL size=14, code size=38]
  63: JIT compiled TypeHandle:AsMethodTable():long:this [Tier0, IL size=7, code size=17]
  64: JIT compiled MethodTable:get_IsValueType():bool:this [Tier0, IL size=20, code
size=32]
  65: JIT compiled GC:KeepAlive(Object) [Tier0, IL size=1, code size=10]
  66: JIT compiled Buffer:_Memmove(byref,byref,long) [Tier0, IL size=25, code size=279]
  67: JIT compiled Environment:InitializeCommandLineArgs(long,int,long):ref [Tier0, IL
size=75, code size=332]
  68: JIT compiled Environment:.cctor() [Tier0, IL size=11, code size=163]
  69: JIT compiled StartupHookProvider:ProcessStartupHooks() [Tier-0 switched to FullOpts,
IL size=365, code size=1053]
  70: JIT compiled StartupHookProvider:get_IsSupported():bool [Tier0, IL size=18, code
size=60]
  71: JIT compiled AppContext:TryGetSwitch(String,byref):bool [Tier0, IL size=97, code
size=322]
  72: JIT compiled ArgumentException:ThrowIfNullOrEmpty(String,String) [Tier0, IL size=16,
code size=53]
  73: JIT compiled String:IsNullOrEmpty(String):bool [Tier0, IL size=15, code size=58]
  74: JIT compiled AppContext:GetData(String):Object [Tier0, IL size=64, code size=205]
  75: JIT compiled ArgumentNullException:ThrowIfNull(Object,String) [Tier0, IL size=10,
code size=42]
  76: JIT compiled Monitor:Enter(Object,byref) [Tier0, IL size=17, code size=55]
```

```
  77: JIT compiled Dictionary`2:TryGetValue(__Canon,byref):bool:this [Tier0, IL size=39,
code size=97]
  78: JIT compiled Dictionary`2:FindValue(__Canon):byref:this [Tier0, IL size=391, code
size=1466]
  79: JIT compiled EventSource:.cctor() [Tier0, IL size=34, code size=80]
  80: JIT compiled EventSource:InitializeIsSupported():bool [Tier0, IL size=18, code
size=60]
  81: JIT compiled RuntimeEventSource:.ctor():this [Tier0, IL size=55, code size=184]
  82: JIT compiled
Guid:.ctor(int,short,short,ubyte,ubyte,ubyte,ubyte,ubyte,ubyte,ubyte,ubyte):this [Tier0, IL
size=86, code size=132]
  83: JIT compiled EventSource:.ctor(Guid,String):this [Tier0, IL size=11, code size=90]
  84: JIT compiled EventSource:.ctor(Guid,String,int,ref):this [Tier0, IL size=58, code
size=187]
  85: JIT compiled EventSource:get_IsSupported():bool [Tier0, IL size=6, code size=11]
  86: JIT compiled TraceLoggingEventHandleTable:.ctor():this [Tier0, IL size=20, code
size=67]
  87: JIT compiled EventSource:ValidateSettings(int):int [Tier0, IL size=37, code size=147]
  88: JIT compiled EventSource:Initialize(Guid,String,ref):this [Tier0, IL size=418, code
size=1584]
  89: JIT compiled Guid:op_Equality(Guid,Guid):bool [Tier0, IL size=10, code size=39]
  90: JIT compiled Guid:EqualsCore(byref,byref):bool [Tier0, IL size=132, code size=171]
  91: JIT compiled ActivityTracker:get_Instance():ActivityTracker [Tier0, IL size=6, code
size=49]
  92: JIT compiled ActivityTracker:.cctor() [Tier0, IL size=11, code size=71]
  93: JIT compiled ActivityTracker:.ctor():this [Tier0, IL size=7, code size=31]
  94: JIT compiled RuntimeEventSource:get_ProviderMetadata():ReadOnlySpan`1:this [Tier0, IL
size=13, code size=91]
  95: JIT compiled ReadOnlySpan`1:.ctor(long,int):this [Tier0, IL size=51, code size=115]
  96: JIT compiled RuntimeHelpers:IsReferenceOrContainsReferences():bool [Tier0, IL size=2,
code size=8]
  97: JIT compiled ReadOnlySpan`1:get_Length():int:this [Tier0, IL size=7, code size=17]
  98: JIT compiled OverrideEventProvider:.ctor(EventSource,int):this [Tier0, IL size=22,
code size=68]
  99: JIT compiled EventProvider:.ctor(int):this [Tier0, IL size=46, code size=194]
 100: JIT compiled EtwEventProvider:.ctor():this [Tier0, IL size=7, code size=31]
 101: JIT compiled EventProvider:Register(EventSource):this [Tier0, IL size=48, code
size=186]
 102: JIT compiled MulticastDelegate:CtorClosed(Object,long):this [Tier0, IL size=23, code
size=70]
 103: JIT compiled EventProvider:EventRegister(EventSource,EtwEnableCallback):int:this
[Tier0, IL size=53, code size=154]
 104: JIT compiled EventSource:get_Name():String:this [Tier0, IL size=7, code size=18]
 105: JIT compiled EventSource:get_Guid():Guid:this [Tier0, IL size=7, code size=41]
 106: JIT compiled
EtwEventProvider:System.Diagnostics.Tracing.IEventProvider.EventRegister(EventSource,EtwEna
bleCallback,long,byref):int:this [Tier0, IL size=19, code size=71]
 107: JIT compiled Advapi32:EventRegister(byref,EtwEnableCallback,long,byref):int [Tier0,
IL size=53, code size=374]
 108: JIT compiled Marshal:GetFunctionPointerForDelegate(__Canon):long [Tier0, IL size=17,
code size=54]
 109: JIT compiled Marshal:GetFunctionPointerForDelegate(Delegate):long [Tier0, IL size=18,
code size=53]
 110: JIT compiled EventPipeEventProvider:.ctor():this [Tier0, IL size=18, code size=41]
 111: JIT compiled EventListener:get_EventListenersLock():Object [Tier0, IL size=41, code
size=157]
 112: JIT compiled List`1:.ctor(int):this [Tier0, IL size=47, code size=275]
 113: JIT compiled Interlocked:CompareExchange(byref,__Canon,__Canon):__Canon [Tier0, IL
size=9, code size=50]
 114: JIT compiled NativeRuntimeEventSource:.cctor() [Tier0, IL size=11, code size=71]
 115: JIT compiled NativeRuntimeEventSource:.ctor():this [Tier0, IL size=63, code size=184]
```

```
 116: JIT compiled
Guid:.ctor(int,ushort,ushort,ubyte,ubyte,ubyte,ubyte,ubyte,ubyte,ubyte,ubyte):this [Tier0,
IL size=88, code size=132]
 117: JIT compiled NativeRuntimeEventSource:get_ProviderMetadata():ReadOnlySpan`1:this
[Tier0, IL size=13, code size=91]
 118: JIT compiled
EventPipeEventProvider:System.Diagnostics.Tracing.IEventProvider.EventRegister(EventSource,
EtwEnableCallback,long,byref):int:this [Tier0, IL size=44, code size=118]
 119: JIT compiled EventPipeInternal:CreateProvider(String,EtwEnableCallback):long [Tier0,
IL size=43, code size=320]
 120: JIT compiled Utf16StringMarshaller:GetPinnableReference(String):byref [Tier0, IL
size=13, code size=50]
 121: JIT compiled String:GetPinnableReference():byref:this [Tier0, IL size=7, code
size=24]
 122: JIT compiled EventListener:AddEventSource(EventSource) [Tier0, IL size=175, code
size=560]
 123: JIT compiled List`1:get_Count():int:this [Tier0, IL size=7, code size=17]
 124: JIT compiled WeakReference`1:.ctor(__Canon):this [Tier0, IL size=9, code size=42]
 125: JIT compiled WeakReference`1:.ctor(__Canon,bool):this [Tier0, IL size=15, code
size=60]
 126: JIT compiled List`1:Add(__Canon):this [Tier0, IL size=60, code size=124]
 127: JIT compiled String:op_Inequality(String,String):bool [Tier0, IL size=11, code
size=46]
 128: JIT compiled String:Equals(String,String):bool [Tier0, IL size=36, code size=114]
 129: JIT compiled ReadOnlySpan`1:GetPinnableReference():byref:this [Tier0, IL size=23,
code size=57]
 130: JIT compiled EventProvider:SetInformation(int,long,int):int:this [Tier0, IL size=38,
code size=131]
 131: JIT compiled ILStubClass:IL_STUB_PInvoke(long,int,long,int):int [FullOpts, IL
size=62, code size=170]
 132: JIT compiled Program:Main() [Tier0, IL size=11, code size=36]
 133: JIT compiled Console:WriteLine(String) [Tier0, IL size=12, code size=59]
 134: JIT compiled Console:get_Out():TextWriter [Tier0, IL size=20, code size=113]
 135: JIT compiled Console:.cctor() [Tier0, IL size=11, code size=71]
 136: JIT compiled Volatile:Read(byref):__Canon [Tier0, IL size=6, code size=21]
 137: JIT compiled Console:<get_Out>g__EnsureInitialized|26_0():TextWriter [Tier0, IL
size=63, code size=209]
 138: JIT compiled ConsolePal:OpenStandardOutput():Stream [Tier0, IL size=34, code
size=130]
 139: JIT compiled Console:get_OutputEncoding():Encoding [Tier0, IL size=72, code size=237]
 140: JIT compiled ConsolePal:get_OutputEncoding():Encoding [Tier0, IL size=11, code
size=200]
 141: JIT compiled NativeLibrary:LoadLibraryCallbackStub(String,Assembly,bool,int):long
[Tier0, IL size=63, code size=280]
 142: JIT compiled EncodingHelper:GetSupportedConsoleEncoding(int):Encoding [Tier0, IL
size=53, code size=186]
 143: JIT compiled Encoding:GetEncoding(int):Encoding [Tier0, IL size=340, code size=1025]
 144: JIT compiled EncodingProvider:GetEncodingFromProvider(int):Encoding [Tier0, IL
size=51, code size=232]
 145: JIT compiled Encoding:FilterDisallowedEncodings(Encoding):Encoding [Tier0, IL
size=29, code size=84]
 146: JIT compiled LocalAppContextSwitches:get_EnableUnsafeUTF7Encoding():bool [Tier0, IL
size=16, code size=46]
 147: JIT compiled LocalAppContextSwitches:GetCachedSwitchValue(String,byref):bool [Tier0,
IL size=22, code size=76]
 148: JIT compiled LocalAppContextSwitches:GetCachedSwitchValueInternal(String,byref):bool
[Tier0, IL size=46, code size=168]
 149: JIT compiled LocalAppContextSwitches:GetSwitchDefaultValue(String):bool [Tier0, IL
size=32, code size=98]
 150: JIT compiled String:op_Equality(String,String):bool [Tier0, IL size=8, code size=39]
 151: JIT compiled Encoding:get_Default():Encoding [Tier0, IL size=6, code size=49]
```

```
 152: JIT compiled Encoding:.cctor() [Tier0, IL size=12, code size=73]
 153: JIT compiled UTF8EncodingSealed:.ctor(bool):this [Tier0, IL size=8, code size=40]
 154: JIT compiled UTF8Encoding:.ctor(bool):this [Tier0, IL size=14, code size=43]
 155: JIT compiled UTF8Encoding:.ctor():this [Tier0, IL size=12, code size=36]
 156: JIT compiled Encoding:.ctor(int):this [Tier0, IL size=42, code size=152]
 157: JIT compiled UTF8Encoding:SetDefaultFallbacks():this [Tier0, IL size=64, code
size=212]
 158: JIT compiled EncoderReplacementFallback:.ctor(String):this [Tier0, IL size=110, code
size=360]
 159: JIT compiled EncoderFallback:.ctor():this [Tier0, IL size=7, code size=31]
 160: JIT compiled String:get_Chars(int):ushort:this [Tier0, IL size=29, code size=61]
 161: JIT compiled Char:IsSurrogate(ushort):bool [Tier0, IL size=17, code size=43]
 162: JIT compiled Char:IsBetween(ushort,ushort,ushort):bool [Tier0, IL size=12, code
size=52]
 163: JIT compiled DecoderReplacementFallback:.ctor(String):this [Tier0, IL size=110, code
size=360]
 164: JIT compiled DecoderFallback:.ctor():this [Tier0, IL size=7, code size=31]
 165: JIT compiled Encoding:get_CodePage():int:this [Tier0, IL size=7, code size=17]
 166: JIT compiled Encoding:get_UTF8():Encoding [Tier0, IL size=6, code size=49]
 167: JIT compiled UTF8Encoding:.cctor() [Tier0, IL size=12, code size=76]
 168: JIT compiled Volatile:Write(byref,__Canon) [Tier0, IL size=6, code size=32]
 169: JIT compiled ConsolePal:GetStandardFile(int,int,bool):Stream [Tier0, IL size=50, code
size=183]
 170: JIT compiled ConsolePal:get_InvalidHandleValue():long [Tier0, IL size=7, code
size=41]
 171: JIT compiled IntPtr:.ctor(int):this [Tier0, IL size=9, code size=25]
 172: JIT compiled ConsolePal:ConsoleHandleIsWritable(long):bool [Tier0, IL size=26, code
size=68]
 173: JIT compiled Kernel32:WriteFile(long,long,int,byref,long):int [Tier0, IL size=46,
code size=294]
 174: JIT compiled Marshal:SetLastSystemError(int) [Tier0, IL size=7, code size=40]
 175: JIT compiled Marshal:GetLastSystemError():int [Tier0, IL size=6, code size=34]
 176: JIT compiled WindowsConsoleStream:.ctor(long,int,bool):this [Tier0, IL size=37, code
size=90]
 177: JIT compiled ConsoleStream:.ctor(int):this [Tier0, IL size=31, code size=71]
 178: JIT compiled Stream:.ctor():this [Tier0, IL size=7, code size=31]
 179: JIT compiled MarshalByRefObject:.ctor():this [Tier0, IL size=7, code size=31]
 180: JIT compiled Kernel32:GetFileType(long):int [Tier0, IL size=27, code size=217]
 181: JIT compiled Console:CreateOutputWriter(Stream):TextWriter [Tier0, IL size=50, code
size=230]
 182: JIT compiled Stream:.cctor() [Tier0, IL size=11, code size=71]
 183: JIT compiled NullStream:.ctor():this [Tier0, IL size=7, code size=31]
 184: JIT compiled EncodingExtensions:RemovePreamble(Encoding):Encoding [Tier0, IL size=25,
code size=118]
 185: JIT compiled UTF8EncodingSealed:get_Preamble():ReadOnlySpan`1:this [Tier0, IL
size=24, code size=99]
 186: JIT compiled UTF8Encoding:get_PreambleSpan():ReadOnlySpan`1 [Tier0, IL size=12, code
size=87]
 187: JIT compiled ConsoleEncoding:.ctor(Encoding):this [Tier0, IL size=14, code size=52]
 188: JIT compiled Encoding:.ctor():this [Tier0, IL size=8, code size=33]
 189: JIT compiled Encoding:SetDefaultFallbacks():this [Tier0, IL size=23, code size=65]
 190: JIT compiled EncoderFallback:get_ReplacementFallback():EncoderFallback [Tier0, IL
size=6, code size=49]
 191: JIT compiled EncoderReplacementFallback:.cctor() [Tier0, IL size=11, code size=71]
 192: JIT compiled EncoderReplacementFallback:.ctor():this [Tier0, IL size=12, code
size=44]
 193: JIT compiled DecoderFallback:get_ReplacementFallback():DecoderFallback [Tier0, IL
size=6, code size=49]
 194: JIT compiled DecoderReplacementFallback:.cctor() [Tier0, IL size=11, code size=71]
 195: JIT compiled DecoderReplacementFallback:.ctor():this [Tier0, IL size=12, code
size=44]
```

```
 196: JIT compiled StreamWriter:.ctor(Stream,Encoding,int,bool):this [Tier0, IL size=201, 
code size=564]
 197: JIT compiled Task:get_CompletedTask():Task [Tier0, IL size=6, code size=49]
 198: JIT compiled Task:.cctor() [Tier0, IL size=76, code size=316]
 199: JIT compiled TaskFactory:.ctor():this [Tier0, IL size=7, code size=31]
 200: JIT compiled Task`1:.ctor(bool,VoidTaskResult,int,CancellationToken):this [Tier0, IL 
size=21, code size=75]
 201: JIT compiled Task:.ctor(bool,int,CancellationToken):this [Tier0, IL size=70, code 
size=181]
 202: JIT compiled <>c:.cctor() [Tier0, IL size=11, code size=71]
 203: JIT compiled <>c:.ctor():this [Tier0, IL size=7, code size=31]
 204: JIT compiled TextWriter:.ctor(IFormatProvider):this [Tier0, IL size=36, code 
size=124]
 205: JIT compiled TextWriter:.cctor() [Tier0, IL size=26, code size=108]
 206: JIT compiled NullTextWriter:.ctor():this [Tier0, IL size=7, code size=31]
 207: JIT compiled TextWriter:.ctor():this [Tier0, IL size=29, code size=103]
 208: JIT compiled String:ToCharArray():ref:this [Tier0, IL size=52, code size=173]
 209: JIT compiled MemoryMarshal:GetArrayDataReference(ref):byref [Tier0, IL size=7, code 
size=24]
 210: JIT compiled ConsoleStream:get_CanWrite():bool:this [Tier0, IL size=7, code size=18]
 211: JIT compiled ConsoleEncoding:GetEncoder():Encoder:this [Tier0, IL size=12, code 
size=57]
 212: JIT compiled UTF8Encoding:GetEncoder():Encoder:this [Tier0, IL size=7, code size=63]
 213: JIT compiled EncoderNLS:.ctor(Encoding):this [Tier0, IL size=37, code size=102]
 214: JIT compiled Encoder:.ctor():this [Tier0, IL size=7, code size=31]
 215: JIT compiled Encoding:get_EncoderFallback():EncoderFallback:this [Tier0, IL size=7, 
code size=18]
 216: JIT compiled EncoderNLS:Reset():this [Tier0, IL size=24, code size=92]
 217: JIT compiled ConsoleStream:get_CanSeek():bool:this [Tier0, IL size=2, code size=12]
 218: JIT compiled StreamWriter:set_AutoFlush(bool):this [Tier0, IL size=25, code size=72]
 219: JIT compiled StreamWriter:CheckAsyncTaskInProgress():this [Tier0, IL size=19, code 
size=47]
 220: JIT compiled Task:get_IsCompleted():bool:this [Tier0, IL size=16, code size=40]
 221: JIT compiled Task:IsCompletedMethod(int):bool [Tier0, IL size=11, code size=25]
 222: JIT compiled StreamWriter:Flush(bool,bool):this [Tier0, IL size=272, code size=1127]
 223: JIT compiled StreamWriter:ThrowIfDisposed():this [Tier0, IL size=15, code size=43]
 224: JIT compiled Encoding:get_Preamble():ReadOnlySpan`1:this [Tier0, IL size=12, code 
size=70]
 225: JIT compiled ConsoleEncoding:GetPreamble():ref:this [Tier0, IL size=6, code size=27]
 226: JIT compiled Array:Empty():ref [Tier0, IL size=6, code size=49]
 227: JIT compiled EmptyArray`1:.cctor() [Tier0, IL size=12, code size=52]
 228: JIT compiled ReadOnlySpan`1:op_Implicit(ref):ReadOnlySpan`1 [Tier0, IL size=7, code 
size=79]
 229: JIT compiled ReadOnlySpan`1:.ctor(ref):this [Tier0, IL size=33, code size=81]
 230: JIT compiled MemoryMarshal:GetArrayDataReference(ref):byref [Tier0, IL size=7, code 
size=24]
 231: JIT compiled ConsoleEncoding:GetMaxByteCount(int):int:this [Tier0, IL size=13, code 
size=63]
 232: JIT compiled UTF8EncodingSealed:GetMaxByteCount(int):int:this [Tier0, IL size=20, 
code size=50]
 233: JIT compiled Span`1:.ctor(long,int):this [Tier0, IL size=51, code size=115]
 234: JIT compiled ReadOnlySpan`1:.ctor(ref,int,int):this [Tier0, IL size=65, code 
size=147]
 235: JIT compiled Encoder:GetBytes(ReadOnlySpan`1,Span`1,bool):int:this [Tier0, IL 
size=44, code size=234]
 236: JIT compiled MemoryMarshal:GetNonNullPinnableReference(ReadOnlySpan`1):byref [Tier0, 
IL size=30, code size=54]
 237: JIT compiled ReadOnlySpan`1:get_Length():int:this [Tier0, IL size=7, code size=17]
 238: JIT compiled MemoryMarshal:GetNonNullPinnableReference(Span`1):byref [Tier0, IL 
size=30, code size=54]
 239: JIT compiled Span`1:get_Length():int:this [Tier0, IL size=7, code size=17]
```

```
 240: JIT compiled EncoderNLS:GetBytes(long,int,long,int,bool):int:this [Tier0, IL size=92,
code size=279]
 241: JIT compiled ArgumentNullException:ThrowIfNull(long,String) [Tier0, IL size=12, code
size=45]
 242: JIT compiled Encoding:GetBytes(long,int,long,int,EncoderNLS):int:this [Tier0, IL
size=57, code size=187]
 243: JIT compiled EncoderNLS:get_HasLeftoverData():bool:this [Tier0, IL size=35, code
size=105]
 244: JIT compiled UTF8Encoding:GetBytesFast(long,int,long,int,byref):int:this [Tier0, IL
size=33, code size=119]
 245: JIT compiled Utf8Utility:TranscodeToUtf8(long,int,long,int,byref,byref):int [Tier0,
IL size=1446, code size=3208]
 246: JIT compiled Math:Min(int,int):int [Tier0, IL size=8, code size=28]
 247: JIT compiled ASCIIUtility:NarrowUtf16ToAscii(long,long,long):long [Tier0, IL
size=490, code size=1187]
 248: JIT compiled WindowsConsoleStream:Flush():this [Tier0, IL size=26, code size=56]
 249: JIT compiled ConsoleStream:Flush():this [Tier0, IL size=1, code size=10]
 250: JIT compiled TextWriter:Synchronized(TextWriter):TextWriter [Tier0, IL size=28, code
size=121]
 251: JIT compiled SyncTextWriter:.ctor(TextWriter):this [Tier0, IL size=14, code size=52]
 252: JIT compiled SyncTextWriter:WriteLine(String):this [Tier0, IL size=13, code size=140]
 253: JIT compiled StreamWriter:WriteLine(String):this [Tier0, IL size=20, code size=110]
 254: JIT compiled String:op_Implicit(String):ReadOnlySpan`1 [Tier0, IL size=31, code
size=171]
 255: JIT compiled String:GetRawStringData():byref:this [Tier0, IL size=7, code size=24]
 256: JIT compiled ReadOnlySpan`1:.ctor(byref,int):this [Tier0, IL size=15, code size=39]
 257: JIT compiled StreamWriter:WriteSpan(ReadOnlySpan`1,bool):this [Tier0, IL size=368,
code size=1036]
 258: JIT compiled MemoryMarshal:GetReference(ReadOnlySpan`1):byref [Tier0, IL size=8, code
size=17]
 259: JIT compiled Buffer:MemoryCopy(long,long,long,long) [Tier0, IL size=21, code size=83]
 260: JIT compiled Unsafe:ReadUnaligned(long):long [Tier0, IL size=10, code size=17]
 261: JIT compiled ASCIIUtility:AllCharsInUInt64AreAscii(long):bool [Tier0, IL size=16,
code size=38]
 262: JIT compiled ASCIIUtility:NarrowFourUtf16CharsToAsciiAndWriteToBuffer(byref,long)
[Tier0, IL size=107, code size=171]
 263: JIT compiled Unsafe:WriteUnaligned(byref,int) [Tier0, IL size=11, code size=22]
 264: JIT compiled Unsafe:ReadUnaligned(long):int [Tier0, IL size=10, code size=16]
 265: JIT compiled ASCIIUtility:AllCharsInUInt32AreAscii(int):bool [Tier0, IL size=11, code
size=25]
 266: JIT compiled ASCIIUtility:NarrowTwoUtf16CharsToAsciiAndWriteToBuffer(byref,int)
[Tier0, IL size=24, code size=35]
 267: JIT compiled Span`1:Slice(int,int):Span`1:this [Tier0, IL size=39, code size=135]
 268: JIT compiled Span`1:.ctor(byref,int):this [Tier0, IL size=15, code size=39]
 269: JIT compiled Span`1:op_Implicit(Span`1):ReadOnlySpan`1 [Tier0, IL size=19, code
size=90]
 270: JIT compiled ReadOnlySpan`1:.ctor(byref,int):this [Tier0, IL size=15, code size=39]
 271: JIT compiled WindowsConsoleStream:Write(ReadOnlySpan`1):this [Tier0, IL size=35, code
size=149]
 272: JIT compiled WindowsConsoleStream:WriteFileNative(long,ReadOnlySpan`1,bool):int
[Tier0, IL size=107, code size=272]
 273: JIT compiled ReadOnlySpan`1:get_IsEmpty():bool:this [Tier0, IL size=10, code size=24]
Hello, world!
 274: JIT compiled AppContext:OnProcessExit() [Tier0, IL size=43, code size=161]
 275: JIT compiled AssemblyLoadContext:OnProcessExit() [Tier0, IL size=101, code size=442]
 276: JIT compiled EventListener:DisposeOnShutdown() [Tier0, IL size=150, code size=618]
 277: JIT compiled List`1:.ctor():this [Tier0, IL size=18, code size=133]
 278: JIT compiled List`1:.cctor() [Tier0, IL size=12, code size=129]
 279: JIT compiled List`1:GetEnumerator():Enumerator:this [Tier0, IL size=7, code size=162]
 280: JIT compiled Enumerator:.ctor(List`1):this [Tier0, IL size=39, code size=64]
 281: JIT compiled Enumerator:MoveNext():bool:this [Tier0, IL size=81, code size=159]
```

```
 282: JIT compiled Enumerator:get_Current():__Canon:this [Tier0, IL size=7, code size=22]
 283: JIT compiled WeakReference`1:TryGetTarget(byref):bool:this [Tier0, IL size=24, code
size=66]
 284: JIT compiled List`1:AddWithResize(__Canon):this [Tier0, IL size=39, code size=85]
 285: JIT compiled List`1:Grow(int):this [Tier0, IL size=53, code size=121]
 286: JIT compiled List`1:set_Capacity(int):this [Tier0, IL size=86, code size=342]
 287: JIT compiled CastHelpers:StelemRef_Helper(byref,long,Object) [Tier0, IL size=34, code
size=104]
 288: JIT compiled CastHelpers:StelemRef_Helper_NoCacheLookup(byref,long,Object) [Tier0, IL
size=26, code size=111]
 289: JIT compiled Enumerator:MoveNextRare():bool:this [Tier0, IL size=57, code size=80]
 290: JIT compiled Enumerator:Dispose():this [Tier0, IL size=1, code size=14]
 291: JIT compiled EventSource:Dispose():this [Tier0, IL size=14, code size=54]
 292: JIT compiled EventSource:Dispose(bool):this [Tier0, IL size=124, code size=236]
 293: JIT compiled EventProvider:Dispose():this [Tier0, IL size=14, code size=54]
 294: JIT compiled EventProvider:Dispose(bool):this [Tier0, IL size=90, code size=230]
 295: JIT compiled EventProvider:EventUnregister(long):this [Tier0, IL size=14, code
size=50]
 296: JIT compiled
EtwEventProvider:System.Diagnostics.Tracing.IEventProvider.EventUnregister(long):int:this
[Tier0, IL size=7, code size=181]
 297: JIT compiled GC:SuppressFinalize(Object) [Tier0, IL size=18, code size=53]
 298: JIT compiled
EventPipeEventProvider:System.Diagnostics.Tracing.IEventProvider.EventUnregister(long):int:
this [Tier0, IL size=13, code size=187]
```

With that out of the way, let's move on to actual performance improvements, starting with on-stack replacement.

On-Stack Replacement

On-stack replacement (OSR) is one of the coolest features to hit the JIT in .NET 7. But to really understand OSR, we first need to understand tiered compilation, so a quick recap…

One of the issues a managed environment with a JIT compiler has to deal with is tradeoffs between startup and throughput. Historically, the job of an optimizing compiler is to, well, optimize, in order to enable the best possible throughput of the application or service once running. But such optimization takes analysis, takes time, and performing all of that work then leads to increased startup time, as all of the code on the startup path (e.g. all of the code that needs to be run before a web server can serve the first request) needs to be compiled. So a JIT compiler needs to make tradeoffs: better throughput at the expense of longer startup time, or better startup time at the expense of decreased throughput. For some kinds of apps and services, the tradeoff is an easy call, e.g. if your service starts up once and then runs for days, several extra seconds of startup time doesn't matter, or if you're a console application that's going to do a quick computation and exit, startup time is all that matters. But how can the JIT know which scenario it's in, and do we really want every developer having to know about these kinds of settings and tradeoffs and configure every one of their applications accordingly? One answer to this has been ahead-of-time compilation, which has taken various forms in .NET. For example, all of the core libraries are "crossgen"'d, meaning they've been run through a tool that produces the previously mentioned R2R format, yielding binaries that contain assembly code that needs only minor tweaks to actually execute; not every method can have code generated for it, but enough that it significantly reduces startup time. Of course, such approaches have their own downsides, e.g. one of the promises of a JIT compiler is it can take advantage of knowledge of the current machine / process in order to best optimize, so for example the R2R images have to assume a certain baseline instruction set (e.g. what vectorizing instructions are available) whereas the JIT can see what's actually available and use the best. "Tiered compilation" provides another answer, one that's usable with or without these other ahead-of-time (AOT) compilation solutions.

Tiered compilation enables the JIT to have its proverbial cake and eat it, too. The idea is simple: allow the JIT to compile the same code multiple times. The first time, the JIT can use as a few optimizations as make sense (a handful of optimizations can actually make the JIT's own throughput faster, so those still make sense to apply), producing fairly unoptimized assembly code but doing so really quickly. And when it does so, it can add some instrumentation into the assembly to track how often the methods are called. As it turns out, many functions used on a startup path are invoked once or maybe only a handful of times, and it would take more time to optimize them than it does to just execute them unoptimized. Then, when the method's instrumentation triggers some threshold, for example a method having been executed 30 times, a work item gets queued to recompile that method, but this time with all the optimizations the JIT can throw at it. This is lovingly referred to as "tiering up." Once that recompilation has completed, call sites to the method are patched with the address of the newly highly optimized assembly code, and future invocations will then take the fast path. So, we get faster startup *and* faster sustained throughput. At least, that's the hope.

A problem, however, is methods that don't fit this mold. While it's certainly the case that many performance-sensitive methods are relatively quick and executed many, many, many times, there's also a large number of performance-sensitive methods that are executed just a handful of times, or maybe even only once, but that take a very long time to execute, maybe even the duration of the whole process: methods with loops. As a result, by default tiered compilation hasn't applied to loops, though it can be enabled by setting the `DOTNET_TC_QuickJitForLoops` environment variable to `1`. We can see the effect of this by trying this simple console app with .NET 6. With the default settings, run this app:

```
class Program
{
    static void Main()
    {
        var sw = new System.Diagnostics.Stopwatch();
        while (true)
        {
            sw.Restart();
            for (int trial = 0; trial < 10_000; trial++)
            {
                int count = 0;
                for (int i = 0; i < char.MaxValue; i++)
                    if (IsAsciiDigit((char)i))
                        count++;
            }
            sw.Stop();
            Console.WriteLine(sw.Elapsed);
        }

        static bool IsAsciiDigit(char c) => (uint)(c - '0') <= 9;
    }
}
```

I get numbers printed out like:

```
00:00:00.5734352
00:00:00.5526667
00:00:00.5675267
00:00:00.5588724
00:00:00.5616028
```

Now, try setting `DOTNET_TC_QuickJitForLoops` to `1`. When I then run it again, I get numbers like this:

```
00:00:01.2841397
00:00:01.2693485
00:00:01.2755646
00:00:01.2656678
00:00:01.2679925
```

In other words, with `DOTNET_TC_QuickJitForLoops` enabled, it's taking 2.5x as long as without (the default in .NET 6). That's because this main function never gets optimizations applied to it. By setting `DOTNET_TC_QuickJitForLoops` to `1`, we're saying "JIT, please apply tiering to methods with loops as well," but this method with a loop is only ever invoked once, so for the duration of the process it ends up remaining at "tier-0," aka unoptimized. Now, let's try the same thing with .NET 7. Regardless of whether that environment variable is set, I again get numbers like this:

```
00:00:00.5528889
00:00:00.5562563
00:00:00.5622086
00:00:00.5668220
00:00:00.5589112
```

but importantly, this method was still participating in tiering. In fact, we can get confirmation of that by using the aforementioned `DOTNET_JitDisasmSummary=1` environment variable. When I set that and run again, I see these lines in the output:

```
   4: JIT compiled Program:Main() [Tier0, IL size=83, code size=319]
...
   6: JIT compiled Program:Main() [Tier1-OSR @0x27, IL size=83, code size=380]
```

highlighting that `Main` was indeed compiled twice. How is that possible? On-stack replacement.

The idea behind on-stack replacement is a method can be replaced not just between invocations but even while it's executing, while it's "on the stack." In addition to the tier-0 code being instrumented for call counts, loops are also instrumented for iteration counts. When the iterations surpass a certain limit, the JIT compiles a new highly optimized version of that method, transfers all the local/register state from the current invocation to the new invocation, and then jumps to the appropriate location in the new method. We can see this in action by using the previously discussed `DOTNET_JitDisasm` environment variable. Set that to `Program:*` in order to see the assembly code generated for all of the methods in the `Program` class, and then run the app again. You should see output like the following:

```
; Assembly listing for method Program:Main()
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-0 compilation
; MinOpts code
; rbp based frame
; partially interruptible

G_M000_IG01:                ;; offset=0000H
       55                   push     rbp
```

```
       4881EC80000000       sub      rsp, 128
       488DAC2480000000     lea      rbp, [rsp+80H]
       C5D857E4             vxorps   xmm4, xmm4
       C5F97F65B0           vmovdqa  xmmword ptr [rbp-50H], xmm4
       33C0                 xor      eax, eax
       488945C0             mov      qword ptr [rbp-40H], rax

G_M000_IG02:                ;; offset=001FH
       48B9002F0B50FC7F0000 mov      rcx, 0x7FFC500B2F00
       E8721FB25F           call     CORINFO_HELP_NEWSFAST
       488945B0             mov      gword ptr [rbp-50H], rax
       488B4DB0             mov      rcx, gword ptr [rbp-50H]
       FF1544C70D00         call     [Stopwatch:.ctor():this]
       488B4DB0             mov      rcx, gword ptr [rbp-50H]
       48894DC0             mov      gword ptr [rbp-40H], rcx
       C745A8E8030000       mov      dword ptr [rbp-58H], 0x3E8

G_M000_IG03:                ;; offset=004BH
       8B4DA8               mov      ecx, dword ptr [rbp-58H]
       FFC9                 dec      ecx
       894DA8               mov      dword ptr [rbp-58H], ecx
       837DA800             cmp      dword ptr [rbp-58H], 0
       7F0E                 jg       SHORT G_M000_IG05

G_M000_IG04:                ;; offset=0059H
       488D4DA8             lea      rcx, [rbp-58H]
       BA06000000           mov      edx, 6
       E8B985AB5F           call     CORINFO_HELP_PATCHPOINT

G_M000_IG05:                ;; offset=0067H
       488B4DC0             mov      rcx, gword ptr [rbp-40H]
       3909                 cmp      dword ptr [rcx], ecx
       FF1585C70D00         call     [Stopwatch:Restart():this]
       33C9                 xor      ecx, ecx
       894DBC               mov      dword ptr [rbp-44H], ecx
       33C9                 xor      ecx, ecx
       894DB8               mov      dword ptr [rbp-48H], ecx
       EB20                 jmp      SHORT G_M000_IG08

G_M000_IG06:                ;; offset=007FH
       8B4DB8               mov      ecx, dword ptr [rbp-48H]
       0FB7C9               movzx    rcx, cx
       FF152DD40B00         call     [Program:<Main>g__IsAsciiDigit|0_0(ushort):bool]
       85C0                 test     eax, eax
       7408                 je       SHORT G_M000_IG07
       8B4DBC               mov      ecx, dword ptr [rbp-44H]
       FFC1                 inc      ecx
       894DBC               mov      dword ptr [rbp-44H], ecx

G_M000_IG07:                ;; offset=0097H
       8B4DB8               mov      ecx, dword ptr [rbp-48H]
       FFC1                 inc      ecx
       894DB8               mov      dword ptr [rbp-48H], ecx

G_M000_IG08:                ;; offset=009FH
       8B4DA8               mov      ecx, dword ptr [rbp-58H]
       FFC9                 dec      ecx
       894DA8               mov      dword ptr [rbp-58H], ecx
       837DA800             cmp      dword ptr [rbp-58H], 0
       7F0E                 jg       SHORT G_M000_IG10
```

```
G_M000_IG09:                ;; offset=00ADH
       488D4DA8             lea      rcx, [rbp-58H]
       BA23000000           mov      edx, 35
       E86585AB5F           call     CORINFO_HELP_PATCHPOINT

G_M000_IG10:                ;; offset=00BBH
       817DB800CA9A3B       cmp      dword ptr [rbp-48H], 0x3B9ACA00
       7CBB                 jl       SHORT G_M000_IG06
       488B4DC0             mov      rcx, gword ptr [rbp-40H]
       3909                 cmp      dword ptr [rcx], ecx
       FF1570C70D00         call     [Stopwatch:get_ElapsedMilliseconds():long:this]
       488BC8               mov      rcx, rax
       FF1507D00D00         call     [Console:WriteLine(long)]
       E96DFFFFFF           jmp      G_M000_IG03

; Total bytes of code 222

; Assembly listing for method Program:<Main>g__IsAsciiDigit|0_0(ushort):bool
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-0 compilation
; MinOpts code
; rbp based frame
; partially interruptible

G_M000_IG01:                ;; offset=0000H
       55                   push     rbp
       488BEC               mov      rbp, rsp
       894D10               mov      dword ptr [rbp+10H], ecx

G_M000_IG02:                ;; offset=0007H
       8B4510               mov      eax, dword ptr [rbp+10H]
       0FB7C0               movzx    rax, ax
       83C0D0               add      eax, -48
       83F809               cmp      eax, 9
       0F96C0               setbe    al
       0FB6C0               movzx    rax, al

G_M000_IG03:                ;; offset=0019H
       5D                   pop      rbp
       C3                   ret
```

A few relevant things to notice here. First, the comments at the top highlight how this code was compiled:

```
; Tier-0 compilation
; MinOpts code
```

So, we know this is the initial version ("Tier-0") of the method compiled with minimal optimization ("MinOpts"). Second, note this line of the assembly:

```
FF152DD40B00         call     [Program:<Main>g__IsAsciiDigit|0_0(ushort):bool]
```

Our `IsAsciiDigit` helper method is trivially inlineable, but it's not getting inlined; instead, the assembly has a call to it, and indeed we can see below the generated code (also "MinOpts") for `IsAsciiDigit`. Why? Because inlining is an optimization (a really important one) that's disabled as part of tier-0 (because the analysis for doing inlining well is also quite costly). Third, we can see the code the JIT is outputting to instrument this method. This is a bit more involved, but I'll point out the relevant parts. First, we see:

```
C745A8E8030000       mov      dword ptr [rbp-58H], 0x3E8
```

That `0x3E8` is the hex value for the decimal 1,000, which is the default number of iterations a loop needs to iterate before the JIT will generate the optimized version of the method (this is configurable via the `DOTNET_TC_OnStackReplacement_InitialCounter` environment variable). So we see 1,000 being stored into this stack location. Then a bit later in the method we see this:

```
G_M000_IG03:                ;; offset=004BH
       8B4DA8               mov      ecx, dword ptr [rbp-58H]
       FFC9                 dec      ecx
       894DA8               mov      dword ptr [rbp-58H], ecx
       837DA800             cmp      dword ptr [rbp-58H], 0
       7F0E                 jg       SHORT G_M000_IG05

G_M000_IG04:                ;; offset=0059H
       488D4DA8             lea      rcx, [rbp-58H]
       BA06000000           mov      edx, 6
       E8B985AB5F           call     CORINFO_HELP_PATCHPOINT

G_M000_IG05:                ;; offset=0067H
```

The generated code is loading that counter into the `ecx` register, decrementing it, storing it back, and then seeing whether the counter dropped to 0. If it didn't, the code skips to `G_M000_IG05`, which is the label for the actual code in the rest of the loop. But if the counter did drop to 0, the JIT proceeds to store relevant state into the the `rcx` and `edx` registers and then calls the `CORINFO_HELP_PATCHPOINT` helper method. That helper is responsible for triggering the creation of the optimized method if it doesn't yet exist, fixing up all appropriate tracking state, and jumping to the new method. And indeed, if you look again at your console output from running the program, you'll see yet another output for the `Main` method:

```
; Assembly listing for method Program:Main()
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-1 compilation
; OSR variant for entry point 0x23
; optimized code
; rsp based frame
; fully interruptible
; No PGO data
; 1 inlinees with PGO data; 8 single block inlinees; 0 inlinees without PGO data

G_M000_IG01:                ;; offset=0000H
       4883EC58             sub      rsp, 88
       4889BC24D8000000     mov      qword ptr [rsp+D8H], rdi
       4889B424D0000000     mov      qword ptr [rsp+D0H], rsi
       48899C24C8000000     mov      qword ptr [rsp+C8H], rbx
       C5F877               vzeroupper
       33C0                 xor      eax, eax
       4889442428           mov      qword ptr [rsp+28H], rax
       4889442420           mov      qword ptr [rsp+20H], rax
       488B9C24A0000000     mov      rbx, gword ptr [rsp+A0H]
       8BBC249C000000       mov      edi, dword ptr [rsp+9CH]
       8BB42498000000       mov      esi, dword ptr [rsp+98H]

G_M000_IG02:                ;; offset=0041H
       EB45                 jmp      SHORT G_M000_IG05
                            align    [0 bytes for IG06]
```

```
G_M000_IG03:                ;; offset=0043H
       33C9                 xor      ecx, ecx
       488B9C24A0000000     mov      rbx, gword ptr [rsp+A0H]
       48894B08             mov      qword ptr [rbx+08H], rcx
       488D4C2428           lea      rcx, [rsp+28H]
       48B87066E68AFD7F0000 mov      rax, 0x7FFD8AE66670

G_M000_IG04:                ;; offset=0060H
       FFD0                 call     rax ; Kernel32:QueryPerformanceCounter(long):int
       488B442428           mov      rax, qword ptr [rsp+28H]
       488B9C24A0000000     mov      rbx, gword ptr [rsp+A0H]
       48894310             mov      qword ptr [rbx+10H], rax
       C6431801             mov      byte  ptr [rbx+18H], 1
       33FF                 xor      edi, edi
       33F6                 xor      esi, esi
       833D92A1E55F00       cmp      dword ptr [(reloc 0x7ffcafe1ae34)], 0
       0F85CA000000         jne      G_M000_IG13

G_M000_IG05:                ;; offset=0088H
       81FE00CA9A3B         cmp      esi, 0x3B9ACA00
       7D17                 jge      SHORT G_M000_IG09

G_M000_IG06:                ;; offset=0090H
       0FB7CE               movzx    rcx, si
       83C1D0               add      ecx, -48
       83F909               cmp      ecx, 9
       7702                 ja       SHORT G_M000_IG08

G_M000_IG07:                ;; offset=009BH
       FFC7                 inc      edi

G_M000_IG08:                ;; offset=009DH
       FFC6                 inc      esi
       81FE00CA9A3B         cmp      esi, 0x3B9ACA00
       7CE9                 jl       SHORT G_M000_IG06

G_M000_IG09:                ;; offset=00A7H
       488B6B08             mov      rbp, qword ptr [rbx+08H]
       48899C24A0000000     mov      gword ptr [rsp+A0H], rbx
       807B1800             cmp      byte  ptr [rbx+18H], 0
       7436                 je       SHORT G_M000_IG12

G_M000_IG10:                ;; offset=00B9H
       488D4C2420           lea      rcx, [rsp+20H]
       48B87066E68AFD7F0000 mov      rax, 0x7FFD8AE66670

G_M000_IG11:                ;; offset=00C8H
       FFD0                 call     rax ; Kernel32:QueryPerformanceCounter(long):int
       488B4C2420           mov      rcx, qword ptr [rsp+20H]
       488B9C24A0000000     mov      rbx, gword ptr [rsp+A0H]
       482B4B10             sub      rcx, qword ptr [rbx+10H]
       4803E9               add      rbp, rcx
       833D2FA1E55F00       cmp      dword ptr [(reloc 0x7ffcafe1ae34)], 0
       48899C24A0000000     mov      gword ptr [rsp+A0H], rbx
       756D                 jne      SHORT G_M000_IG14

G_M000_IG12:                ;; offset=00EFH
       C5F857C0             vxorps   xmm0, xmm0
       C4E1FB2AC5           vcvtsi2sd  xmm0, rbp
       C5FB11442430         vmovsd   qword ptr [rsp+30H], xmm0
```

```
       48B9F04BF24FFC7F0000 mov      rcx, 0x7FFC4FF24BF0
       BAE7070000           mov      edx, 0x7E7
       E82E1FB25F           call     CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
       C5FB10442430         vmovsd   xmm0, qword ptr [rsp+30H]
       C5FB5905E049F6FF     vmulsd   xmm0, xmm0, qword ptr [(reloc 0x7ffc4ff25720)]
       C4E1FB2CD0           vcvttsd2si  rdx, xmm0
       48B94B598638D6C56D34 mov      rcx, 0x346DC5D63886594B
       488BC1               mov      rax, rcx
       48F7EA               imul     rdx:rax, rdx
       488BCA               mov      rcx, rdx
       48C1E93F             shr      rcx, 63
       48C1FA0B             sar      rdx, 11
       4803CA               add      rcx, rdx
       FF1567CE0D00         call     [Console:WriteLine(long)]
       E9F5FEFFFF           jmp      G_M000_IG03

G_M000_IG13:                ;; offset=014EH
       E8DDCBAC5F           call     CORINFO_HELP_POLL_GC
       E930FFFFFF           jmp      G_M000_IG05

G_M000_IG14:                ;; offset=0158H
       E8D3CBAC5F           call     CORINFO_HELP_POLL_GC
       EB90                 jmp      SHORT G_M000_IG12

; Total bytes of code 351
```

Here, again, we notice a few interesting things. First, in the header we see this:

```
; Tier-1 compilation
; OSR variant for entry point 0x23
; optimized code
```

so we know this is both optimized “tier-1” code and is the “OSR variant” for this method. Second, notice there’s no longer a call to the `IsAsciiDigit` helper. Instead, where that call would have been, we see this:

```
G_M000_IG06:                ;; offset=0090H
       0FB7CE               movzx    rcx, si
       83C1D0               add      ecx, -48
       83F909               cmp      ecx, 9
       7702                 ja       SHORT G_M000_IG08
```

This is loading a value into `rcx`, subtracting 48 from it (48 is the decimal ASCII value of the `'0'` character) and comparing the resulting value to 9. Sounds an awful lot like our `IsAsciiDigit` implementation (`(uint)(c - '0') <= 9`), doesn’t it? That’s because it is. The helper was successfully inlined in this now-optimized code.

Great, so now in .NET 7, we can largely avoid the tradeoffs between startup and throughput, as OSR enables tiered compilation to apply to all methods, even those that are long-running. A multitude of PRs went into enabling this, including many over the last few years, but all of the functionality was disabled in the shipping bits. Thanks to improvements like [dotnet/runtime#62831](https://github.com/dotnet/runtime/pull/62831) which implemented support for OSR on Arm64 (previously only x64 support was implemented), and [dotnet/runtime#63406](https://github.com/dotnet/runtime/pull/63406) and [dotnet/runtime#65609](https://github.com/dotnet/runtime/pull/65609) which revised how OSR imports and epilogs are handled, [dotnet/runtime#65675](https://github.com/dotnet/runtime/pull/65675) enables OSR (and as a result `DOTNET_TC_QuickJitForLoops`) by default.

But, tiered compilation and OSR aren't just about startup (though they're of course very valuable there). They're also about further improving throughput. Even though tiered compilation was originally envisioned as a way to optimize startup while not hurting throughput, it's become much more than that. There are various things the JIT can learn about a method during tier-0 that it can then use for tier-1. For example, the very fact that the tier-0 code executed means that any `statics` accessed by the method will have been initialized, and that means that any `readonly statics` will not only have been initialized by the time the tier-1 code executes but their values won't ever change. And that in turn means that any readonly statics of primitive types (e.g. `bool`, `int`, etc.) can be treated like `consts` instead of `static readonly` fields, and during tier-1 compilation the JIT can optimize them just as it would have optimized a `const`. For example, try running this simple program after setting `DOTNET_JitDisasm` to `Program:Test`:

```
using System.Runtime.CompilerServices;

class Program
{
    static readonly bool Is64Bit = Environment.Is64BitProcess;

    static int Main()
    {
        int count = 0;
        for (int i = 0; i < 1_000_000_000; i++)
            if (Test())
                count++;
        return count;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    static bool Test() => Is64Bit;
}
```

When I do so, I get this output:

```
; Assembly listing for method Program:Test():bool
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-0 compilation
; MinOpts code
; rbp based frame
; partially interruptible

G_M000_IG01:                ;; offset=0000H
       55                   push     rbp
       4883EC20             sub      rsp, 32
       488D6C2420           lea      rbp, [rsp+20H]

G_M000_IG02:                ;; offset=000AH
       48B9B8639A3FFC7F0000 mov      rcx, 0x7FFC3F9A63B8
       BA01000000           mov      edx, 1
       E8C220B25F           call     CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
       0FB60545580C00       movzx    rax, byte  ptr [(reloc 0x7ffc3f9a63ea)]

G_M000_IG03:                ;; offset=0025H
       4883C420             add      rsp, 32
       5D                   pop      rbp
       C3                   ret

; Total bytes of code 43
```

```
; Assembly listing for method Program:Test():bool
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-1 compilation
; optimized code
; rsp based frame
; partially interruptible
; No PGO data

G_M000_IG01:                ;; offset=0000H

G_M000_IG02:                ;; offset=0000H
       B801000000           mov      eax, 1

G_M000_IG03:                ;; offset=0005H
       C3                   ret

; Total bytes of code 6
```

Note, again, we see two outputs for `Program:Test`. First, we see the "Tier-0" code, which is accessing a static (note the `call      CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE` instruction). But then we see the "Tier-1" code, where all of that overhead has vanished and is instead replaced simply by `mov eax, 1`. Since the "Tier-0" code had to have executed in order for it to tier up, the "Tier-1" code was generated knowing that the value of the `static readonly bool Is64Bit` field was `true` (`1`), and so the entirety of this method is storing the value `1` into the `eax` register used for the return value.

This is so useful that components are now written with tiering in mind. Consider the new `Regex` source generator, which is discussed later in this post (Roslyn source generators were introduced a couple of years ago; just as how Roslyn analyzers are able to plug into the compiler and surface additional diagnostics based on all of the data the compiler learns from the source code, Roslyn source generators are able to analyze that same data and then further augment the compilation unit with additional source). The `Regex` source generator applies a technique based on this in [dotnet/runtime#67775](). `Regex` supports setting a process-wide timeout that gets applied to `Regex` instances that don't explicitly set a timeout. That means, even though it's super rare for such a process-wide timeout to be set, the `Regex` source generator still needs to output timeout-related code just in case it's needed. It does so by outputting some helpers like this:

```
static class Utilities
{
    internal static readonly TimeSpan s_defaultTimeout =
AppContext.GetData("REGEX_DEFAULT_MATCH_TIMEOUT") is TimeSpan timeout ? timeout :
Timeout.InfiniteTimeSpan;
    internal static readonly bool s_hasTimeout = s_defaultTimeout !=
Timeout.InfiniteTimeSpan;
}
```

which it then uses at call sites like this:

```
if (Utilities.s_hasTimeout)
{
    base.CheckTimeout();
}
```

In tier-0, these checks will still be emitted in the assembly code, but in tier-1 where throughput matters, if the relevant `AppContext` switch hasn't been set, then `s_defaultTimeout` will be

`Timeout.InfiniteTimeSpan`, at which point `s_hasTimeout` will be `false`. And since `s_hasTimeout` is a `static readonly bool`, the JIT will be able to treat that as a `const`, and all conditions like `if (Utilities.s_hasTimeout)` will be treated equal to `if (false)` and be eliminated from the assembly code entirely as dead code.

But, this is somewhat old news. The JIT has been able to do such an optimization since tiered compilation was introduced in .NET Core 3.0. Now in .NET 7, though, with OSR it's also able to do so by default for methods with loops (and thus enable cases like the regex one). However, the real magic of OSR comes into play when combined with another exciting feature: dynamic PGO.

PGO

I wrote about profile-guided optimization (PGO) in my [Performance Improvements in .NET 6](https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/) post, but I'll cover it again here as it's seen a multitude of improvements for .NET 7.

PGO has been around for a long time, in any number of languages and compilers. The basic idea is you compile your app, asking the compiler to inject instrumentation into the application to track various pieces of interesting information. You then put your app through its paces, running through various common scenarios, causing that instrumentation to "profile" what happens when the app is executed, and the results of that are then saved out. The app is then recompiled, feeding those instrumentation results back into the compiler, and allowing it to optimize the app for exactly how it's expected to be used. This approach to PGO is referred to as "static PGO," as the information is all gleaned ahead of actual deployment, and it's something .NET has been doing in various forms for years. From my perspective, though, the really interesting development in .NET is "dynamic PGO," which was introduced in .NET 6, but off by default.

Dynamic PGO takes advantage of tiered compilation. I noted that the JIT instruments the tier-0 code to track how many times the method is called, or in the case of loops, how many times the loop executes. It can instrument it for other things as well. For example, it can track exactly which concrete types are used as the target of an interface dispatch, and then in tier-1 specialize the code to expect the most common types (this is referred to as "guarded devirtualization," or GDV). You can see this in this little example. Set the `DOTNET_TieredPGO` environment variable to `1`, and then run this on .NET 7:

```
class Program
{
    static void Main()
    {
        IPrinter printer = new Printer();
        for (int i = 0; ; i++)
        {
            DoWork(printer, i);
        }
    }

    static void DoWork(IPrinter printer, int i)
    {
        printer.PrintIfTrue(i == int.MaxValue);
    }

    interface IPrinter
    {
        void PrintIfTrue(bool condition);
```

```
    }

    class Printer : IPrinter
    {
        public void PrintIfTrue(bool condition)
        {
            if (condition) Console.WriteLine("Print!");
        }
    }
}
```

The tier-0 code for `DoWork` ends up looking like this:

```
G_M000_IG01:                ;; offset=0000H
       55                   push     rbp
       4883EC30             sub      rsp, 48
       488D6C2430           lea      rbp, [rsp+30H]
       33C0                 xor      eax, eax
       488945F8             mov      qword ptr [rbp-08H], rax
       488945F0             mov      qword ptr [rbp-10H], rax
       48894D10             mov      gword ptr [rbp+10H], rcx
       895518               mov      dword ptr [rbp+18H], edx

G_M000_IG02:                ;; offset=001BH
       FF059F220F00         inc      dword ptr [(reloc 0x7ffc3f1b2ea0)]
       488B4D10             mov      rcx, gword ptr [rbp+10H]
       48894DF8             mov      gword ptr [rbp-08H], rcx
       488B4DF8             mov      rcx, gword ptr [rbp-08H]
       48BAA82E1B3FFC7F0000 mov      rdx, 0x7FFC3F1B2EA8
       E8B47EC55F           call     CORINFO_HELP_CLASSPROFILE32
       488B4DF8             mov      rcx, gword ptr [rbp-08H]
       48894DF0             mov      gword ptr [rbp-10H], rcx
       488B4DF0             mov      rcx, gword ptr [rbp-10H]
       33D2                 xor      edx, edx
       817D18FFFFFF7F       cmp      dword ptr [rbp+18H], 0x7FFFFFFF
       0F94C2               sete     dl
       49BB0800F13EFC7F0000 mov      r11, 0x7FFC3EF10008
       41FF13               call     [r11]IPrinter:PrintIfTrue(bool):this
       90                   nop

G_M000_IG03:                ;; offset=0062H
       4883C430             add      rsp, 48
       5D                   pop      rbp
       C3                   ret
```

and most notably, you can see the `call [r11]IPrinter:PrintIfTrue(bool):this` doing the interface dispatch. But, then look at the code generated for tier-1. We still see the `call [r11]IPrinter:PrintIfTrue(bool):this`, *but* we also see this:

```
G_M000_IG02:                ;; offset=0020H
       48B9982D1B3FFC7F0000 mov      rcx, 0x7FFC3F1B2D98
       48390F               cmp      qword ptr [rdi], rcx
       7521                 jne      SHORT G_M000_IG05
       81FEFFFFFF7F         cmp      esi, 0x7FFFFFFF
       7404                 je       SHORT G_M000_IG04

G_M000_IG03:                ;; offset=0037H
       FFC6                 inc      esi
       EBE5                 jmp      SHORT G_M000_IG02
```

```
G_M000_IG04:                ;; offset=003BH
       48B9D820801A24020000 mov      rcx, 0x2241A8020D8
       488B09               mov      rcx, gword ptr [rcx]
       FF1572CD0D00         call     [Console:WriteLine(String)]
       EBE7                 jmp      SHORT G_M000_IG03
```

That first block is checking the concrete type of the `IPrinter` (stored in `rdi`) and comparing it against the known type for `Printer` (`0x7FFC3F1B2D98`). If they're different, it just jumps to the same interface dispatch it was doing in the unoptimized version. But if they're the same, it then jumps directly to an inlined version of `Printer.PrintIfTrue` (you can see the call to `Console:WriteLine` right there in this method). Thus, the common case (the only case in this example) is super efficient at the expense of a single comparison and branch.

That all existed in .NET 6, so why are we talking about it now? Several things have improved. First, PGO now works with OSR, thanks to improvements like [dotnet/runtime#61453](). That's a big deal, as it means hot long-running methods that do this kind of interface dispatch (which are fairly common) can get these kinds of devirtualization/inlining optimizations. Second, while PGO isn't currently enabled by default, we've made it much easier to turn on. Between [dotnet/runtime#71438]() and [dotnet/sdk#26350](), it's now possible to simply put `<TieredPGO>true</TieredPGO>` into your .csproj, and it'll have the same effect as if you set `DOTNET_TieredPGO=1` prior to every invocation of the app, enabling dynamic PGO (note that it *doesn't* disable use of R2R images, so if you want the entirety of the core libraries also employing dynamic PGO, you'll also need to set `DOTNET_ReadyToRun=0`). Third, however, is dynamic PGO has been taught how to instrument and optimize additional things.

PGO already knew how to instrument virtual dispatch. Now in .NET 7, thanks in large part to [dotnet/runtime#68703](), it can do so for delegates as well (at least for delegates to instance methods). Consider this simple console app:

```csharp
using System.Runtime.CompilerServices;

class Program
{
    static int[] s_values = Enumerable.Range(0, 1_000).ToArray();

    static void Main()
    {
        for (int i = 0; i < 1_000_000; i++)
            Sum(s_values, i => i * 42);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    static int Sum(int[] values, Func<int, int> func)
    {
        int sum = 0;
        foreach (int value in values)
            sum += func(value);
        return sum;
    }
}
```

Without PGO enabled, I get generated optimized assembly like this:

```
; Assembly listing for method Program:Sum(ref,Func`2):int
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-1 compilation
```

```
; optimized code
; rsp based frame
; partially interruptible
; No PGO data

G_M000_IG01:                ;; offset=0000H
       4156                 push     r14
       57                   push     rdi
       56                   push     rsi
       55                   push     rbp
       53                   push     rbx
       4883EC20             sub      rsp, 32
       488BF2               mov      rsi, rdx

G_M000_IG02:                ;; offset=000DH
       33FF                 xor      edi, edi
       488BD9               mov      rbx, rcx
       33ED                 xor      ebp, ebp
       448B7308             mov      r14d, dword ptr [rbx+08H]
       4585F6               test     r14d, r14d
       7E16                 jle      SHORT G_M000_IG04

G_M000_IG03:                ;; offset=001DH
       8BD5                 mov      edx, ebp
       8B549310             mov      edx, dword ptr [rbx+4*rdx+10H]
       488B4E08             mov      rcx, gword ptr [rsi+08H]
       FF5618               call     [rsi+18H]Func`2:Invoke(int):int:this
       03F8                 add      edi, eax
       FFC5                 inc      ebp
       443BF5               cmp      r14d, ebp
       7FEA                 jg       SHORT G_M000_IG03

G_M000_IG04:                ;; offset=0033H
       8BC7                 mov      eax, edi

G_M000_IG05:                ;; offset=0035H
       4883C420             add      rsp, 32
       5B                   pop      rbx
       5D                   pop      rbp
       5E                   pop      rsi
       5F                   pop      rdi
       415E                 pop      r14
       C3                   ret

; Total bytes of code 64
```

Note the `call [rsi+18H]Func'2:Invoke(int):int:this` in there that's invoking the delegate. Now with PGO enabled:

```
; Assembly listing for method Program:Sum(ref,Func`2):int
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-1 compilation
; optimized code
; optimized using profile data
; rsp based frame
; fully interruptible
; with Dynamic PGO: edge weights are valid, and fgCalledCount is 5628
; 0 inlinees with PGO data; 1 single block inlinees; 0 inlinees without PGO data

G_M000_IG01:                ;; offset=0000H
```

```
        4157                push      r15
        4156                push      r14
        57                  push      rdi
        56                  push      rsi
        55                  push      rbp
        53                  push      rbx
        4883EC28            sub       rsp, 40
        488BF2              mov       rsi, rdx

G_M000_IG02:                ;; offset=000FH
        33FF                xor       edi, edi
        488BD9              mov       rbx, rcx
        33ED                xor       ebp, ebp
        448B7308            mov       r14d, dword ptr [rbx+08H]
        4585F6              test      r14d, r14d
        7E27                jle       SHORT G_M000_IG05

G_M000_IG03:                ;; offset=001FH
        8BC5                mov       eax, ebp
        8B548310            mov       edx, dword ptr [rbx+4*rax+10H]
        4C8B4618            mov       r8, qword ptr [rsi+18H]
        48B8A0C2CF3CFC7F0000 mov      rax, 0x7FFC3CCFC2A0
        4C3BC0              cmp       r8, rax
        751D                jne       SHORT G_M000_IG07
        446BFA2A            imul      r15d, edx, 42

G_M000_IG04:                ;; offset=003CH
        4103FF              add       edi, r15d
        FFC5                inc       ebp
        443BF5              cmp       r14d, ebp
        7FD9                jg        SHORT G_M000_IG03

G_M000_IG05:                ;; offset=0046H
        8BC7                mov       eax, edi

G_M000_IG06:                ;; offset=0048H
        4883C428            add       rsp, 40
        5B                  pop       rbx
        5D                  pop       rbp
        5E                  pop       rsi
        5F                  pop       rdi
        415E                pop       r14
        415F                pop       r15
        C3                  ret

G_M000_IG07:                ;; offset=0055H
        488B4E08            mov       rcx, gword ptr [rsi+08H]
        41FFD0              call      r8
        448BF8              mov       r15d, eax
        EBDB                jmp       SHORT G_M000_IG04
```

I chose the `42` constant in `i => i * 42` to make it easy to see in the assembly, and sure enough, there it is:

```
G_M000_IG03:                ;; offset=001FH
        8BC5                mov       eax, ebp
        8B548310            mov       edx, dword ptr [rbx+4*rax+10H]
        4C8B4618            mov       r8, qword ptr [rsi+18H]
        48B8A0C2CF3CFC7F0000 mov      rax, 0x7FFC3CCFC2A0
        4C3BC0              cmp       r8, rax
```

```
    751D                jne      SHORT G_M000_IG07
    446BFA2A            imul     r15d, edx, 42
```

This is loading the target address from the delegate into `r8` and is loading the address of the expected target into `rax`. If they're the same, it then simply performs the inlined operation (`imul r15d, edx, 42`), and otherwise it jumps to G_M000_IG07 which calls to the function in `r8`. The effect of this is obvious if we run this as a benchmark:

```
static int[] s_values = Enumerable.Range(0, 1_000).ToArray();

[Benchmark]
public int DelegatePGO() => Sum(s_values, i => i * 42);

static int Sum(int[] values, Func<int, int>? func)
{
    int sum = 0;
    foreach (int value in values)
    {
        sum += func(value);
    }
    return sum;
}
```

With PGO disabled, we get the same performance throughput for .NET 6 and .NET 7:

Method	Runtime	Mean	Ratio
DelegatePGO	.NET 6.0	1.665 us	1.00
DelegatePGO	.NET 7.0	1.659 us	1.00

But the picture changes when we enable dynamic PGO (`DOTNET_TieredPGO=1`). .NET 6 gets ~14% faster, but .NET 7 gets ~3x faster!

Method	Runtime	Mean	Ratio
DelegatePGO	.NET 6.0	1,427.7 ns	1.00
DelegatePGO	.NET 7.0	539.0 ns	0.38

[dotnet/runtime#70377](https://github.com/dotnet/runtime/pull/70377) is another valuable improvement with dynamic PGO, which enables PGO to play nicely with loop cloning and invariant hoisting. To understand this better, a brief digression into what those are. Loop cloning is a mechanism the JIT employs to avoid various overheads in the fast path of a loop. Consider the `Test` method in this example:

```
using System.Runtime.CompilerServices;

class Program
{
    static void Main()
    {
        int[] array = new int[10_000_000];
        for (int i = 0; i < 1_000_000; i++)
        {
            Test(array);
        }
    }
```

```
    [MethodImpl(MethodImplOptions.NoInlining)]
    private static bool Test(int[] array)
    {
        for (int i = 0; i < 0x12345; i++)
        {
            if (array[i] == 42)
            {
                return true;
            }
        }

        return false;
    }
}
```

The JIT doesn't know whether the passed in array is of sufficient length that all accesses to `array[i]` inside the loop will be in bounds, and thus it would need to inject bounds checks for every access. While it'd be nice to simply do the length check up front and simply throw an exception early if it wasn't long enough, doing so could also change behavior (imagine the method were writing into the array as it went, or otherwise mutating some shared state). Instead, the JIT employs "loop cloning." It essentially rewrites this `Test` method to be more like this:

```
if (array is not null && array.Length >= 0x12345)
{
    for (int i = 0; i < 0x12345; i++)
    {
        if (array[i] == 42) // no bounds checks emitted for this access :-)
        {
            return true;
        }
    }
}
else
{
    for (int i = 0; i < 0x12345; i++)
    {
        if (array[i] == 42) // bounds checks emitted for this access :-(
        {
            return true;
        }
    }
}
return false;
```

That way, at the expense of some code duplication, we get our fast loop without bounds checks and only pay for the bounds checks in the slow path. You can see this in the generated assembly (if you can't already tell, `DOTNET_JitDisasm` is one of my favorite features in .NET 7):

```
; Assembly listing for method Program:Test(ref):bool
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-1 compilation
; optimized code
; rsp based frame
; fully interruptible
; No PGO data

G_M000_IG01:                ;; offset=0000H
       4883EC28             sub      rsp, 40
```

```
G_M000_IG02:                ;; offset=0004H
       33C0                 xor      eax, eax
       4885C9               test     rcx, rcx
       7429                 je       SHORT G_M000_IG05
       81790845230100       cmp      dword ptr [rcx+08H], 0x12345
       7C20                 jl       SHORT G_M000_IG05
       0F1F40000F1F840000000000 align    [12 bytes for IG03]

G_M000_IG03:                ;; offset=0020H
       8BD0                 mov      edx, eax
       837C91102A           cmp      dword ptr [rcx+4*rdx+10H], 42
       7429                 je       SHORT G_M000_IG08
       FFC0                 inc      eax
       3D45230100           cmp      eax, 0x12345
       7CEE                 jl       SHORT G_M000_IG03

G_M000_IG04:                ;; offset=0032H
       EB17                 jmp      SHORT G_M000_IG06

G_M000_IG05:                ;; offset=0034H
       3B4108               cmp      eax, dword ptr [rcx+08H]
       7323                 jae      SHORT G_M000_IG10
       8BD0                 mov      edx, eax
       837C91102A           cmp      dword ptr [rcx+4*rdx+10H], 42
       7410                 je       SHORT G_M000_IG08
       FFC0                 inc      eax
       3D45230100           cmp      eax, 0x12345
       7CE9                 jl       SHORT G_M000_IG05

G_M000_IG06:                ;; offset=004BH
       33C0                 xor      eax, eax

G_M000_IG07:                ;; offset=004DH
       4883C428             add      rsp, 40
       C3                   ret

G_M000_IG08:                ;; offset=0052H
       B801000000           mov      eax, 1

G_M000_IG09:                ;; offset=0057H
       4883C428             add      rsp, 40
       C3                   ret

G_M000_IG10:                ;; offset=005CH
       E81FA0C15F           call     CORINFO_HELP_RNGCHKFAIL
       CC                   int3

; Total bytes of code 98
```

That G_M000_IG02 section is doing the null check and the length check, jumping to the G_M000_IG05 block if either fails. If both succeed, it's then executing the loop (block G_M000_IG03) without bounds checks:

```
G_M000_IG03:                ;; offset=0020H
       8BD0                 mov      edx, eax
       837C91102A           cmp      dword ptr [rcx+4*rdx+10H], 42
       7429                 je       SHORT G_M000_IG08
       FFC0                 inc      eax
```

```
        3D45230100           cmp      eax, 0x12345
        7CEE                 jl       SHORT G_M000_IG03
```

with the bounds checks only showing up in the slow-path block:

```
G_M000_IG05:                 ;; offset=0034H
        3B4108               cmp      eax, dword ptr [rcx+08H]
        7323                 jae      SHORT G_M000_IG10
        8BD0                 mov      edx, eax
        837C91102A           cmp      dword ptr [rcx+4*rdx+10H], 42
        7410                 je       SHORT G_M000_IG08
        FFC0                 inc      eax
        3D45230100           cmp      eax, 0x12345
        7CE9                 jl       SHORT G_M000_IG05
```

That's "loop cloning." What about "invariant hoisting"? Hoisting means pulling something out of a loop to be before the loop, and invariants are things that don't change. Thus invariant hoisting is pulling something out of a loop to before the loop in order to avoid recomputing every iteration of the loop an answer that won't change. Effectively, the previous example already showed invariant hoisting, in that the bounds check is moved to be before the loop rather than in the loop, but a more concrete example would be something like this:

```
[MethodImpl(MethodImplOptions.NoInlining)]
private static bool Test(int[] array)
{
    for (int i = 0; i < 0x12345; i++)
    {
        if (array[i] == array.Length - 42)
        {
            return true;
        }
    }

    return false;
}
```

Note that the value of `array.Length - 42` doesn't change on each iteration of the loop, so it's "invariant" to the loop iteration and can be lifted out, which the generated code does:

```
G_M000_IG02:                 ;; offset=0004H
        33D2                 xor      edx, edx
        4885C9               test     rcx, rcx
        742A                 je       SHORT G_M000_IG05
        448B4108             mov      r8d, dword ptr [rcx+08H]
        4181F845230100       cmp      r8d, 0x12345
        7C1D                 jl       SHORT G_M000_IG05
        4183C0D6             add      r8d, -42
        0F1F4000             align    [4 bytes for IG03]

G_M000_IG03:                 ;; offset=0020H
        8BC2                 mov      eax, edx
        4439448110           cmp      dword ptr [rcx+4*rax+10H], r8d
        7433                 je       SHORT G_M000_IG08
        FFC2                 inc      edx
        81FA45230100         cmp      edx, 0x12345
        7CED                 jl       SHORT G_M000_IG03
```

Here again we see the array being tested for null (`test rcx, rcx`) and the array's length being checked (`mov r8d, dword ptr [rcx+08H]`, `cmp r8d, 0x12345`), but then with the array's length in `r8d`, we then see this up-front block subtracting 42 from the length (`add r8d, -42`), and that's before we continue into the fast-path loop in the G_M000_IG03 block. This keeps that additional set of operations out of the loop, thereby avoiding the overhead of recomputing the value per iteration.

Ok, so how does this apply to dynamic PGO? Remember that with the interface/virtual dispatch avoidance PGO is able to do, it does so by doing a type check to see whether the type in use is the most common type; if it is, it uses a fast path that calls directly to that type's method (and in doing so that call is then potentially inlined), and if it isn't, it falls back to normal interface/virtual dispatch. That check can be invariant to a loop. So when a method is tiered up and PGO kicks in, the type check can now be hoisted out of the loop, making it even cheaper to handle the common case. Consider this variation of our original example:

```
using System.Runtime.CompilerServices;

class Program
{
    static void Main()
    {
        IPrinter printer = new BlankPrinter();
        while (true)
        {
            DoWork(printer);
        }
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    static void DoWork(IPrinter printer)
    {
        for (int j = 0; j < 123; j++)
        {
            printer.Print(j);
        }
    }

    interface IPrinter
    {
        void Print(int i);
    }

    class BlankPrinter : IPrinter
    {
        public void Print(int i)
        {
            Console.Write("");
        }
    }
}
```

When we look at the optimized assembly generated for this with dynamic PGO enabled, we see this:

```
; Assembly listing for method Program:DoWork(IPrinter)
; Emitting BLENDED_CODE for X64 CPU with AVX - Windows
; Tier-1 compilation
; optimized code
; optimized using profile data
```

```
; rsp based frame
; partially interruptible
; with Dynamic PGO: edge weights are invalid, and fgCalledCount is 12187
; 0 inlinees with PGO data; 1 single block inlinees; 0 inlinees without PGO data

G_M000_IG01:                ;; offset=0000H
       57                   push     rdi
       56                   push     rsi
       4883EC28             sub      rsp, 40
       488BF1               mov      rsi, rcx

G_M000_IG02:                ;; offset=0009H
       33FF                 xor      edi, edi
       4885F6               test     rsi, rsi
       742B                 je       SHORT G_M000_IG05
       48B9982DD43CFC7F0000 mov      rcx, 0x7FFC3CD42D98
       48390E               cmp      qword ptr [rsi], rcx
       751C                 jne      SHORT G_M000_IG05

G_M000_IG03:                ;; offset=001FH
       48B9282040F948020000 mov      rcx, 0x248F9402028
       488B09               mov      rcx, gword ptr [rcx]
       FF1526A80D00         call     [Console:Write(String)]
       FFC7                 inc      edi
       83FF7B               cmp      edi, 123
       7CE6                 jl       SHORT G_M000_IG03

G_M000_IG04:                ;; offset=0039H
       EB29                 jmp      SHORT G_M000_IG07

G_M000_IG05:                ;; offset=003BH
       48B9982DD43CFC7F0000 mov      rcx, 0x7FFC3CD42D98
       48390E               cmp      qword ptr [rsi], rcx
       7521                 jne      SHORT G_M000_IG08
       48B9282040F948020000 mov      rcx, 0x248F9402028
       488B09               mov      rcx, gword ptr [rcx]
       FF15FBA70D00         call     [Console:Write(String)]

G_M000_IG06:                ;; offset=005DH
       FFC7                 inc      edi
       83FF7B               cmp      edi, 123
       7CD7                 jl       SHORT G_M000_IG05

G_M000_IG07:                ;; offset=0064H
       4883C428             add      rsp, 40
       5E                   pop      rsi
       5F                   pop      rdi
       C3                   ret

G_M000_IG08:                ;; offset=006BH
       488BCE               mov      rcx, rsi
       8BD7                 mov      edx, edi
       49BB1000AA3CFC7F0000 mov      r11, 0x7FFC3CAA0010
       41FF13               call     [r11]IPrinter:Print(int):this
       EBDE                 jmp      SHORT G_M000_IG06

; Total bytes of code 127
```

We can see in the G_M000_IG02 block that it's doing the type check on the `IPrinter` instance and jumping to G_M000_IG05 if the check fails (`mov rcx, 0x7FFC3CD42D98`, `cmp qword ptr [rsi], rcx`, `jne SHORT G_M000_IG05`), otherwise falling through to G_M000_IG03 which is a tight fast-path loop that's inlined `BlankPrinter.Print` with no type checks in sight!

Interestingly, improvements like this can bring with them their own challenges. PGO leads to a significant increase in the number of type checks, since call sites that specialize for a given type need to compare against that type. However, common subexpression elimination (CSE) hasn't historically worked for such type handles (CSE is a compiler optimization where duplicate expressions are eliminated by computing the result once and then storing it for subsequent use rather than recomputing it each time). [dotnet/runtime#70580](https://github.com/dotnet/runtime/pull/70580) fixes this by enabling CSE for such constant handles. For example, consider this method:

```
[Benchmark]
[Arguments("", "", "", "")]
public bool AllAreStrings(object o1, object o2, object o3, object o4) =>
    o1 is string && o2 is string && o3 is string && o4 is string;
```

On .NET 6, the JIT produced this assembly code:

```
; Program.AllAreStrings(System.Object, System.Object, System.Object, System.Object)
       test      rdx,rdx
       je        short M00_L01
       mov       rax,offset MT_System.String
       cmp       [rdx],rax
       jne       short M00_L01
       test      r8,r8
       je        short M00_L01
       mov       rax,offset MT_System.String
       cmp       [r8],rax
       jne       short M00_L01
       test      r9,r9
       je        short M00_L01
       mov       rax,offset MT_System.String
       cmp       [r9],rax
       jne       short M00_L01
       mov       rax,[rsp+28]
       test      rax,rax
       je        short M00_L00
       mov       rdx,offset MT_System.String
       cmp       [rax],rdx
       je        short M00_L00
       xor       eax,eax
M00_L00:
       test      rax,rax
       setne     al
       movzx     eax,al
       ret
M00_L01:
       xor       eax,eax
       ret
; Total bytes of code 100
```

Note the C# has four tests for `string` and the assembly code has four loads with `mov rax,offset MT_System.String`. Now on .NET 7, the load is performed just once:

```
; Program.AllAreStrings(System.Object, System.Object, System.Object, System.Object)
       test      rdx,rdx
       je        short M00_L01
```

```
        mov       rax,offset MT_System.String
        cmp       [rdx],rax
        jne       short M00_L01
        test      r8,r8
        je        short M00_L01
        cmp       [r8],rax
        jne       short M00_L01
        test      r9,r9
        je        short M00_L01
        cmp       [r9],rax
        jne       short M00_L01
        mov       rdx,[rsp+28]
        test      rdx,rdx
        je        short M00_L00
        cmp       [rdx],rax
        je        short M00_L00
        xor       edx,edx
M00_L00:
        xor       eax,eax
        test      rdx,rdx
        setne     al
        ret
M00_L01:
        xor       eax,eax
        ret
; Total bytes of code 69
```

Bounds Check Elimination

One of the things that makes .NET attractive is its safety. The runtime guards access to arrays, strings, and spans such that you can't accidentally corrupt memory by walking off either end; if you do, rather than reading/writing arbitrary memory, you'll get exceptions. Of course, that's not magic; it's done by the JIT inserting bounds checks every time one of these data structures is indexed. For example, this:

```
[MethodImpl(MethodImplOptions.NoInlining)]
static int Read0thElement(int[] array) => array[0];
```

results in:

```
G_M000_IG01:                ;; offset=0000H
       4883EC28             sub      rsp, 40

G_M000_IG02:                ;; offset=0004H
       83790800             cmp      dword ptr [rcx+08H], 0
       7608                 jbe      SHORT G_M000_IG04
       8B4110               mov      eax, dword ptr [rcx+10H]

G_M000_IG03:                ;; offset=000DH
       4883C428             add      rsp, 40
       C3                   ret

G_M000_IG04:                ;; offset=0012H
       E8E9A0C25F           call     CORINFO_HELP_RNGCHKFAIL
       CC                   int3
```

The array is passed into this method in the `rcx` register, pointing to the method table pointer in the object, and the length of an array is stored in the object just after that method table pointer (which is 8 bytes in a 64-bit process). Thus the `cmp dword ptr [rcx+08H], 0` instruction is reading the length of the array and comparing the length to 0; that makes sense, since the length can't be negative, and we're trying to access the 0th element, so as long as the length isn't 0, the array has enough elements for us to access its 0th element. In the event that the length was 0, the code jumps to the end of the function, which contains `call CORINFO_HELP_RNGCHKFAIL`; that's a JIT helper function that throws an `IndexOutOfRangeException`. If the length was sufficient, however, it then reads the `int` stored at the beginning of the array's data, which on 64-bit is 16 bytes (0x10) past the pointer (`mov eax, dword ptr [rcx+10H]`).

While these bounds checks in and of themselves aren't super expensive, do a lot of them and their costs add up. So while the JIT needs to ensure that "safe" accesses don't go out of bounds, it also tries to prove that certain accesses won't, in which case it needn't emit the bounds check that it knows will be superfluous. In every release of .NET, more and more cases have been added to find places these bounds checks can be eliminated, and .NET 7 is no exception.

For example, [dotnet/runtime#61662](https://github.com/dotnet/runtime/pull/61662) from [@anthonycanino](https://github.com/anthonycanino) enabled the JIT to understand various forms of binary operations as part of range checks. Consider this method:

```
[MethodImpl(MethodImplOptions.NoInlining)]
private static ushort[]? Convert(ReadOnlySpan<byte> bytes)
{
    if (bytes.Length != 16)
    {
        return null;
    }

    var result = new ushort[8];
    for (int i = 0; i < result.Length; i++)
    {
        result[i] = (ushort)(bytes[i * 2] * 256 + bytes[i * 2 + 1]);
    }

    return result;
}
```

It's validating that the input span is 16 bytes long and then creating a `new ushort[8]` where each `ushort` in the array combines two of the input bytes. To do that, it's looping over the output array, and indexing into the bytes array using `i * 2` and `i * 2 + 1` as the indices. On .NET 6, each of those indexing operations would result in a bounds check, with assembly like:

```
cmp       r8d,10
jae       short G_M000_IG04
movsxd    r8,r8d
```

where that G_M000_IG04 is the `call CORINFO_HELP_RNGCHKFAIL` we're now familiar with. But on .NET 7, we get this assembly for the method:

```
G_M000_IG01:                ;; offset=0000H
       56                   push     rsi
       4883EC20             sub      rsp, 32

G_M000_IG02:                ;; offset=0005H
       488B31               mov      rsi, bword ptr [rcx]
       8B4908               mov      ecx, dword ptr [rcx+08H]
```

```
       83F910               cmp      ecx, 16
       754C                 jne      SHORT G_M000_IG05
       48B9302F542FFC7F0000 mov      rcx, 0x7FFC2F542F30
       BA08000000           mov      edx, 8
       E80C1EB05F           call     CORINFO_HELP_NEWARR_1_VC
       33D2                 xor      edx, edx
                            align    [0 bytes for IG03]

G_M000_IG03:                ;; offset=0026H
       8D0C12               lea      ecx, [rdx+rdx]
       448BC1               mov      r8d, ecx
       FFC1                 inc      ecx
       458BC0               mov      r8d, r8d
       460FB60406           movzx    r8, byte  ptr [rsi+r8]
       41C1E008             shl      r8d, 8
       8BC9                 mov      ecx, ecx
       0FB60C0E             movzx    rcx, byte  ptr [rsi+rcx]
       4103C8               add      ecx, r8d
       0FB7C9               movzx    rcx, cx
       448BC2               mov      r8d, edx
       6642894C4010         mov      word  ptr [rax+2*r8+10H], cx
       FFC2                 inc      edx
       83FA08               cmp      edx, 8
       7CD0                 jl       SHORT G_M000_IG03

G_M000_IG04:                ;; offset=0056H
       4883C420             add      rsp, 32
       5E                   pop      rsi
       C3                   ret

G_M000_IG05:                ;; offset=005CH
       33C0                 xor      rax, rax

G_M000_IG06:                ;; offset=005EH
       4883C420             add      rsp, 32
       5E                   pop      rsi
       C3                   ret

; Total bytes of code 100
```

No bounds checks, which is most easily seen by the lack of the telltale `call CORINFO_HELP_RNGCHKFAIL` at the end of the method. With this PR, the JIT is able to understand the impact of certain multiplication and shift operations and their relationships to the bounds of the data structure. Since it can see that the result array's length is 8 and the loop is iterating from 0 to that exclusive upper bound, it knows that `i` will always be in the range `[0, 7]`, which means that `i * 2` will always be in the range `[0, 14]` and `i * 2 + 1` will always be in the range `[0, 15]`. As such, it's able to prove that the bounds checks aren't needed.

[dotnet/runtime#61569](#) and [dotnet/runtime#62864](#) also help to eliminate bounds checks when dealing with constant strings and spans initialized from RVA statics ("Relative Virtual Address" static fields, basically a static field that lives in a module's data section). For example, consider this benchmark:

```
[Benchmark]
[Arguments(1)]
public char GetChar(int i)
{
    const string Text = "hello";
```

```
    return (uint)i < Text.Length ? Text[i] : '\0';
}
```

On .NET 6, we get this assembly:

```
; Program.GetChar(Int32)
       sub       rsp,28
       mov       eax,edx
       cmp       rax,5
       jl        short M00_L00
       xor       eax,eax
       add       rsp,28
       ret
M00_L00:
       cmp       edx,5
       jae       short M00_L01
       mov       rax,2278B331450
       mov       rax,[rax]
       movsxd    rdx,edx
       movzx     eax,word ptr [rax+rdx*2+0C]
       add       rsp,28
       ret
M00_L01:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 56
```

The beginning of this makes sense: the JIT was obviously able to see that the length of `Text` is 5, so it's implementing the `(uint)i < Text.Length` check by doing `cmp rax,5`, and if `i` as an unsigned value is greater than or equal to 5, it's then zero'ing out the return value (to return the `'\0'`) and exiting. If the length is less than 5 (in which case it's also at least 0 due to the unsigned comparison), it then jumps to M00_L00 to read the value from the string… but we then see another `cmp` against 5, this time as part of a range check. So even though the JIT knew the index was in bounds, it wasn't able to remove the bounds check. Now it is; in .NET 7, we get this:

```
; Program.GetChar(Int32)
       cmp       edx,5
       jb        short M00_L00
       xor       eax,eax
       ret
M00_L00:
       mov       rax,2B0AF002530
       mov       rax,[rax]
       mov       edx,edx
       movzx     eax,word ptr [rax+rdx*2+0C]
       ret
; Total bytes of code 29
```

So much nicer.

dotnet/runtime#67141 is a great example of how evolving ecosystem needs drives specific optimizations into the JIT. The `Regex` compiler and source generator handle some cases of regular expression character classes by using a bitmap lookup stored in strings. For example, to determine whether a `char c` is in the character class `"[A-Za-z0-9_]"` (which will match an underscore or any ASCII letter or digit), the implementation ends up generating an expression like the body of the following method:

```
[Benchmark]
[Arguments('a')]
public bool IsInSet(char c) =>
    c < 128 && ("\0\0\0\u03FF\uFFFE\u87FF\uFFFE\u07FF"[c >> 4] & (1 << (c & 0xF))) != 0;
```

The implementation is treating an 8-character string as a 128-bit lookup table. If the character is known to be in range (such that it's effectively a 7-bit value), it's then using the top 3 bits of the value to index into the 8 elements of the string, and the bottom 4 bits to select one of the 16 bits in that element, giving us an answer as to whether this input character is in the set or not. In .NET 6, even though we know the character is in range of the string, the JIT couldn't see through either the length comparison or the bit shift.

```
; Program.IsInSet(Char)
       sub       rsp,28
       movzx     eax,dx
       cmp       eax,80
       jge       short M00_L00
       mov       edx,eax
       sar       edx,4
       cmp       edx,8
       jae       short M00_L01
       mov       rcx,299835A1518
       mov       rcx,[rcx]
       movsxd    rdx,edx
       movzx     edx,word ptr [rcx+rdx*2+0C]
       and       eax,0F
       bt        edx,eax
       setb      al
       movzx     eax,al
       add       rsp,28
       ret
M00_L00:
       xor       eax,eax
       add       rsp,28
       ret
M00_L01:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 75
```

The previously mentioned PR takes care of the length check. And this PR takes care of the bit shift. So in .NET 7, we get this loveliness:

```
; Program.IsInSet(Char)
       movzx     eax,dx
       cmp       eax,80
       jge       short M00_L00
       mov       edx,eax
       sar       edx,4
       mov       rcx,197D4800608
       mov       rcx,[rcx]
       mov       edx,edx
       movzx     edx,word ptr [rcx+rdx*2+0C]
       and       eax,0F
       bt        edx,eax
       setb      al
       movzx     eax,al
       ret
```

```
M00_L00:
       xor       eax,eax
       ret
; Total bytes of code 51
```

Note the distinct lack of a `call CORINFO_HELP_RNGCHKFAIL`. And as you might guess, this check can happen *a lot* in a `Regex`, making this a very useful addition.

Bounds checks are an obvious source of overhead when talking about array access, but they're not the only ones. There's also the need to use the cheapest instructions possible. In .NET 6, with a method like:

```
[MethodImpl(MethodImplOptions.NoInlining)]
private static int Get(int[] values, int i) => values[i];
```

assembly code like the following would be generated:

```
; Program.Get(Int32[], Int32)
       sub       rsp,28
       cmp       edx,[rcx+8]
       jae       short M01_L00
       movsxd    rax,edx
       mov       eax,[rcx+rax*4+10]
       add       rsp,28
       ret
M01_L00:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 27
```

This should look fairly familiar from our previous discussion; the JIT is loading the array's length (`[rcx+8]`) and comparing that with the value of `i` (in `edx`), and then jumping to the end to throw an exception if `i` is out of bounds. Immediately after that jump we see a `movsxd rax, edx` instruction, which is taking the 32-bit value of `i` from `edx` and moving it into the 64-bit register `rax`. And as part of moving it, it's sign-extending it; that's the "sxd" part of the instruction name (sign-extending means the upper 32 bits of the new 64-bit value will be set to the value of the upper bit of the 32-bit value, so that the number retains its signed value). The interesting thing is, though, we know that the `Length` of an array and of a span is non-negative, and since we just bounds checked `i` against the `Length`, we also know that `i` is non-negative. That makes such sign-extension useless, since the upper bit is guaranteed to be 0. Since the `mov` instruction that zero-extends is a tad cheaper than `movsxd`, we can simply use that instead. And that's exactly what [dotnet/runtime#57970](https://github.com/dotnet/runtime/pull/57970) from [@pentp](https://github.com/pentp) does for both arrays and spans ([dotnet/runtime#70884](https://github.com/dotnet/runtime/pull/70884) also similarly avoids some signed casts in other situations). Now on .NET 7, we get this:

```
; Program.Get(Int32[], Int32)
       sub       rsp,28
       cmp       edx,[rcx+8]
       jae       short M01_L00
       mov       eax,edx
       mov       eax,[rcx+rax*4+10]
       add       rsp,28
       ret
M01_L00:
       call      CORINFO_HELP_RNGCHKFAIL
```

```
        int        3
; Total bytes of code 26
```

That's not the only source of overhead with array access, though. In fact, there's a very large category of array access overhead that's been there forever, but that's so well known there are even old FxCop rules and newer Roslyn analyzers that warn against it: multidimensional array accesses. The overhead in the case of a multidimensional array isn't just an extra branch on every indexing operation, or additional math required to compute the location of the element, but rather that they currently pass through the JIT's optimization phases largely unmodified. [dotnet/runtime#70271](https://github.com/dotnet/runtime/pull/70271) improves the state of the world here by doing an expansion of a multidimensional array access early in the JIT's pipeline, such that later optimization phases can improve multidimensional accesses as they would other code, including CSE and loop invariant hoisting. The impact of this is visible in a simple benchmark that sums all the elements of a multidimensional array.

```
private int[,] _square;

[Params(1000)]
public int Size { get; set; }

[GlobalSetup]
public void Setup()
{
    int count = 0;
    _square = new int[Size, Size];
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
        {
            _square[i, j] = count++;
        }
    }
}

[Benchmark]
public int Sum()
{
    int[,] square = _square;
    int sum = 0;
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
        {
            sum += square[i, j];
        }
    }
    return sum;
}
```

Method	Runtime	Mean	Ratio
Sum	.NET 6.0	964.1 us	1.00
Sum	.NET 7.0	674.7 us	0.70

This previous example assumes you know the size of each dimension of the multidimensional array (it's referring to the `Size` directly in the loops). That's obviously not always (or maybe even rarely) the case. In such situations, you'd be more likely to use the `Array.GetUpperBound` method, and because multidimensional arrays can have a non-zero lower bound, `Array.GetLowerBound`. That would lead to code like this:

```
private int[,] _square;

[Params(1000)]
public int Size { get; set; }

[GlobalSetup]
public void Setup()
{
    int count = 0;
    _square = new int[Size, Size];
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
        {
            _square[i, j] = count++;
        }
    }
}

[Benchmark]
public int Sum()
{
    int[,] square = _square;
    int sum = 0;
    for (int i = square.GetLowerBound(0); i < square.GetUpperBound(0); i++)
    {
        for (int j = square.GetLowerBound(1); j < square.GetUpperBound(1); j++)
        {
            sum += square[i, j];
        }
    }
    return sum;
}
```

In .NET 7, thanks to [dotnet/runtime#60816](https://github.com/dotnet/runtime/pull/60816), those `GetLowerBound` and `GetUpperBound` calls become JIT intrinsics. An "intrinsic" to a compiler is something the compiler has intrinsic knowledge of, such that rather than relying solely on a method's defined implementation (if it even has one), the compiler can substitute in something it considers to be better. There are literally thousands of methods in .NET known in this manner to the JIT, with `GetLowerBound` and `GetUpperBound` being two of the most recent. Now as intrinsics, when they're passed a constant value (e.g. `0` for the 0th rank), the JIT can substitute the necessary assembly instructions to read directly from the memory location that houses the bounds. Here's what the assembly code for this benchmark looked like with .NET 6; the main thing to see here are all of the `call`s out to `GetLowerBound` and `GetUpperBound`:

```
; Program.Sum()
       push      rdi
       push      rsi
       push      rbp
       push      rbx
       sub       rsp,28
       mov       rsi,[rcx+8]
       xor       edi,edi
       mov       rcx,rsi
       xor       edx,edx
```

```
        cmp       [rcx],ecx
        call      System.Array.GetLowerBound(Int32)
        mov       ebx,eax
        mov       rcx,rsi
        xor       edx,edx
        call      System.Array.GetUpperBound(Int32)
        cmp       eax,ebx
        jle       short M00_L03
M00_L00:
        mov       rcx,[rsi]
        mov       ecx,[rcx+4]
        add       ecx,0FFFFFFE8
        shr       ecx,3
        cmp       ecx,1
        jbe       short M00_L05
        lea       rdx,[rsi+10]
        inc       ecx
        movsxd    rcx,ecx
        mov       ebp,[rdx+rcx*4]
        mov       rcx,rsi
        mov       edx,1
        call      System.Array.GetUpperBound(Int32)
        cmp       eax,ebp
        jle       short M00_L02
M00_L01:
        mov       ecx,ebx
        sub       ecx,[rsi+18]
        cmp       ecx,[rsi+10]
        jae       short M00_L04
        mov       edx,ebp
        sub       edx,[rsi+1C]
        cmp       edx,[rsi+14]
        jae       short M00_L04
        mov       eax,[rsi+14]
        imul      rax,rcx
        mov       rcx,rdx
        add       rcx,rax
        add       edi,[rsi+rcx*4+20]
        inc       ebp
        mov       rcx,rsi
        mov       edx,1
        call      System.Array.GetUpperBound(Int32)
        cmp       eax,ebp
        jg        short M00_L01
M00_L02:
        inc       ebx
        mov       rcx,rsi
        xor       edx,edx
        call      System.Array.GetUpperBound(Int32)
        cmp       eax,ebx
        jg        short M00_L00
M00_L03:
        mov       eax,edi
        add       rsp,28
        pop       rbx
        pop       rbp
        pop       rsi
        pop       rdi
        ret
M00_L04:
        call      CORINFO_HELP_RNGCHKFAIL
```

```
M00_L05:
       mov      rcx,offset MT_System.IndexOutOfRangeException
       call     CORINFO_HELP_NEWSFAST
       mov      rsi,rax
       call     System.SR.get_IndexOutOfRange_ArrayRankIndex()
       mov      rdx,rax
       mov      rcx,rsi
       call     System.IndexOutOfRangeException..ctor(System.String)
       mov      rcx,rsi
       call     CORINFO_HELP_THROW
       int      3
; Total bytes of code 219
```

Now here's what it is for .NET 7:

```
; Program.Sum()
       push     r14
       push     rdi
       push     rsi
       push     rbp
       push     rbx
       sub      rsp,20
       mov      rdx,[rcx+8]
       xor      eax,eax
       mov      ecx,[rdx+18]
       mov      r8d,ecx
       mov      r9d,[rdx+10]
       lea      ecx,[rcx+r9+0FFFF]
       cmp      ecx,r8d
       jle      short M00_L03
       mov      r9d,[rdx+1C]
       mov      r10d,[rdx+14]
       lea      r10d,[r9+r10+0FFFF]
M00_L00:
       mov      r11d,r9d
       cmp      r10d,r11d
       jle      short M00_L02
       mov      esi,r8d
       sub      esi,[rdx+18]
       mov      edi,[rdx+10]
M00_L01:
       mov      ebx,esi
       cmp      ebx,edi
       jae      short M00_L04
       mov      ebp,[rdx+14]
       imul     ebx,ebp
       mov      r14d,r11d
       sub      r14d,[rdx+1C]
       cmp      r14d,ebp
       jae      short M00_L04
       add      ebx,r14d
       add      eax,[rdx+rbx*4+20]
       inc      r11d
       cmp      r10d,r11d
       jg       short M00_L01
M00_L02:
       inc      r8d
       cmp      ecx,r8d
       jg       short M00_L00
M00_L03:
```

```
        add       rsp,20
        pop       rbx
        pop       rbp
        pop       rsi
        pop       rdi
        pop       r14
        ret
M00_L04:
        call      CORINFO_HELP_RNGCHKFAIL
        int       3
; Total bytes of code 130
```

Importantly, note there are no more `calls` (other than for the bounds check exception at the end). For example, instead of that first `GetUpperBound` call:

```
call      System.Array.GetUpperBound(Int32)
```

we get:

```
mov       r9d,[rdx+1C]
mov       r10d,[rdx+14]
lea       r10d,[r9+r10+0FFFF]
```

and it ends up being much faster:

Method	Runtime	Mean	Ratio
Sum	.NET 6.0	2,657.5 us	1.00
Sum	.NET 7.0	676.3 us	0.25

Loop Hoisting and Cloning

We previously saw how PGO interacts with loop hoisting and cloning, and those optimizations have seen other improvements, as well.

Historically, the JIT's support for hoisting has been limited to lifting an invariant out one level. Consider this example:

```
[Benchmark]
public void Compute()
{
    for (int thousands = 0; thousands < 10; thousands++)
    {
        for (int hundreds = 0; hundreds < 10; hundreds++)
        {
            for (int tens = 0; tens < 10; tens++)
            {
                for (int ones = 0; ones < 10; ones++)
                {
                    int n = ComputeNumber(thousands, hundreds, tens, ones);
                    Process(n);
                }
            }
        }
    }
}
```

```
static int ComputeNumber(int thousands, int hundreds, int tens, int ones) =>
    (thousands * 1000) +
    (hundreds * 100) +
    (tens * 10) +
    ones;

[MethodImpl(MethodImplOptions.NoInlining)]
static void Process(int n) { }
```

At first glance, you might look at this and say "what could be hoisted, the computation of `n` requires all of the loop inputs, and all of that computation is in `ComputeNumber`." But from a compiler's perspective, the `ComputeNumber` function is inlineable and thus logically can be part of its caller, the computation of `n` is actually split into multiple pieces, and each of those pieces can be hoisted to different levels, e.g. the tens computation can be hoisted out one level, the hundreds out two levels, and the thousands out three levels. Here's what `[DisassemblyDiagnoser]` outputs for .NET 6:

```
; Program.Compute()
       push      r14
       push      rdi
       push      rsi
       push      rbp
       push      rbx
       sub       rsp,20
       xor       esi,esi
M00_L00:
       xor       edi,edi
M00_L01:
       xor       ebx,ebx
M00_L02:
       xor       ebp,ebp
       imul      ecx,esi,3E8
       imul      eax,edi,64
       add       ecx,eax
       lea       eax,[rbx+rbx*4]
       lea       r14d,[rcx+rax*2]
M00_L03:
       lea       ecx,[r14+rbp]
       call      Program.Process(Int32)
       inc       ebp
       cmp       ebp,0A
       jl        short M00_L03
       inc       ebx
       cmp       ebx,0A
       jl        short M00_L02
       inc       edi
       cmp       edi,0A
       jl        short M00_L01
       inc       esi
       cmp       esi,0A
       jl        short M00_L00
       add       rsp,20
       pop       rbx
       pop       rbp
       pop       rsi
       pop       rdi
       pop       r14
       ret
; Total bytes of code 84
```

We can see that *some* hoisting has happened here. After all, the inner most loop (tagged M00_L03) is only five instructions: increment `ebp` (which at this point is the `ones` counter value), and if it's still less than 0xA (10), jump back to M00_L03 which adds whatever is in `r14` to `ones`. Great, so we've hoisted all of the unnecessary computation out of the inner loop, being left only with adding the ones position to the rest of the number. Let's go out a level. M00_L02 is the label for the tens loop. What do we see there? Trouble. The two instructions `imul ecx,esi,3E8` and `imul eax,edi,64` are performing the `thousands * 1000` and `hundreds * 100` operations, highlighting that these operations which could have been hoisted out further were left stuck in the next-to-innermost loop. Now, here's what we get for .NET 7, where this was improved in dotnet/runtime#68061:

```
; Program.Compute()
       push      r15
       push      r14
       push      r12
       push      rdi
       push      rsi
       push      rbp
       push      rbx
       sub       rsp,20
       xor       esi,esi
M00_L00:
       xor       edi,edi
       imul      ebx,esi,3E8
M00_L01:
       xor       ebp,ebp
       imul      r14d,edi,64
       add       r14d,ebx
M00_L02:
       xor       r15d,r15d
       lea       ecx,[rbp+rbp*4]
       lea       r12d,[r14+rcx*2]
M00_L03:
       lea       ecx,[r12+r15]
       call      qword ptr [Program.Process(Int32)]
       inc       r15d
       cmp       r15d,0A
       jl        short M00_L03
       inc       ebp
       cmp       ebp,0A
       jl        short M00_L02
       inc       edi
       cmp       edi,0A
       jl        short M00_L01
       inc       esi
       cmp       esi,0A
       jl        short M00_L00
       add       rsp,20
       pop       rbx
       pop       rbp
       pop       rsi
       pop       rdi
       pop       r12
       pop       r14
       pop       r15
       ret
; Total bytes of code 99
```

Notice now where those `imul` instructions live. There are four labels, each one corresponding to one of the loops, and we can see the outermost loop has the `imul ebx,esi,3E8` (for the thousands computation) and the next loop has the `imul r14d,edi,64` (for the hundreds computation), highlighting that these computations were hoisted out to the appropriate level (the tens and ones computation are still in the right places).

More improvements have gone in on the cloning side. Previously, loop cloning would only apply for loops iterating by 1 from a low to a high value. With [dotnet/runtime#60148](https://github.com/dotnet/runtime/pull/60148), the comparison against the upper value can be `<=` rather than just `<`. And with [dotnet/runtime#67930](https://github.com/dotnet/runtime/pull/67930), loops that iterate downward can also be cloned, as can loops that have increments and decrements larger than 1. Consider this benchmark:

```
private int[] _values = Enumerable.Range(0, 1000).ToArray();

[Benchmark]
[Arguments(0, 0, 1000)]
public int LastIndexOf(int arg, int offset, int count)
{
    int[] values = _values;
    for (int i = offset + count - 1; i >= offset; i--)
        if (values[i] == arg)
            return i;
    return 0;
}
```

Without loop cloning, the JIT can't assume that `offset` through `offset+count` are in range, and thus every access to the array needs to be bounds checked. With loop cloning, the JIT could generate one version of the loop without bounds checks and only use that when it knows all accesses will be valid. That's exactly what happens now in .NET 7. Here's what we got with .NET 6:

```
; Program.LastIndexOf(Int32, Int32, Int32)
       sub       rsp,28
       mov       rcx,[rcx+8]
       lea       eax,[r8+r9+0FFFF]
       cmp       eax,r8d
       jl        short M00_L01
       mov       r9d,[rcx+8]
       nop       word ptr [rax+rax]
M00_L00:
       cmp       eax,r9d
       jae       short M00_L03
       movsxd    r10,eax
       cmp       [rcx+r10*4+10],edx
       je        short M00_L02
       dec       eax
       cmp       eax,r8d
       jge       short M00_L00
M00_L01:
       xor       eax,eax
       add       rsp,28
       ret
M00_L02:
       add       rsp,28
       ret
M00_L03:
       call      CORINFO_HELP_RNGCHKFAIL
```

```
        int       3
; Total bytes of code 72
```

Notice how in the core loop, at label M00_L00, there's a bounds check (`cmp eax,r9d` and `jae short M00_L03`, which jumps to a `call CORINFO_HELP_RNGCHKFAIL`). And here's what we get with .NET 7:

```
; Program.LastIndexOf(Int32, Int32, Int32)
       sub       rsp,28
       mov       rax,[rcx+8]
       lea       ecx,[r8+r9+0FFFF]
       cmp       ecx,r8d
       jl        short M00_L02
       test      rax,rax
       je        short M00_L01
       test      ecx,ecx
       jl        short M00_L01
       test      r8d,r8d
       jl        short M00_L01
       cmp       [rax+8],ecx
       jle       short M00_L01
M00_L00:
       mov       r9d,ecx
       cmp       [rax+r9*4+10],edx
       je        short M00_L03
       dec       ecx
       cmp       ecx,r8d
       jge       short M00_L00
       jmp       short M00_L02
M00_L01:
       cmp       ecx,[rax+8]
       jae       short M00_L04
       mov       r9d,ecx
       cmp       [rax+r9*4+10],edx
       je        short M00_L03
       dec       ecx
       cmp       ecx,r8d
       jge       short M00_L01
M00_L02:
       xor       eax,eax
       add       rsp,28
       ret
M00_L03:
       mov       eax,ecx
       add       rsp,28
       ret
M00_L04:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 98
```

Notice how the code size is larger, and how there are now two variations of the loop: one at M00_L00 and one at M00_L01. The second one, M00_L01, has a branch to that same `call CORINFO_HELP_RNGCHKFAIL`, but the first one doesn't, because that loop will only end up being used after proving that the `offset`, `count`, and `_values.Length` are such that the indexing will always be in bounds.

Other changes also improved loop cloning. [dotnet/runtime#59886](https://github.com/dotnet/runtime/pull/59886) enables the JIT to choose different forms for how to emit the the conditions for choosing the fast or slow loop path, e.g. whether to emit all the conditions, `&` them together, and then branch (`if (!(cond1 & cond2)) goto slowPath`), or whether to emit each condition on its own (`if (!cond1) goto slowPath; if (!cond2) goto slowPath`). [dotnet/runtime#66257](https://github.com/dotnet/runtime/pull/66257) enables loop cloning to kick in when the loop variable is initialized to more kinds of expressions (e.g. `for (int fromindex = lastIndex - lengthToClear; ...)`). And [dotnet/runtime#70232](https://github.com/dotnet/runtime/pull/70232) increases the JIT's willingness to clone loops with bodies that do a broader set of operations.

Folding, propagation, and substitution

Constant folding is an optimization where a compiler computes the value of an expression involving only constants at compile-time rather than generating the code to compute the value at run-time. There are multiple levels of constant folding in .NET, with some constant folding performed by the C# compiler and some constant folding performed by the JIT compiler. For example, given the C# code:

```
[Benchmark]
public int A() => 3 + (4 * 5);

[Benchmark]
public int B() => A() * 2;
```

the C# compiler will generate IL for these methods like the following:

```
.method public hidebysig instance int32 A () cil managed
{
    .maxstack 8
    IL_0000: ldc.i4.s 23
    IL_0002: ret
}

.method public hidebysig instance int32 B () cil managed
{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call instance int32 Program::A()
    IL_0006: ldc.i4.2
    IL_0007: mul
    IL_0008: ret
}
```

You can see that the C# compiler has computed the value of `3 + (4*5)`, as the IL for method `A` simply contains the equivalent of `return 23;`. However, method `B` contains the equivalent of `return A() * 2;`, highlighting that the constant folding performed by the C# compiler was intramethod only. Now here's what the JIT generates:

```
; Program.A()
       mov       eax,17
       ret
; Total bytes of code 6

; Program.B()
       mov       eax,2E
       ret
; Total bytes of code 6
```

The assembly for method `A` isn't particularly interesting; it's just returning that same value 23 (hex 0x17). But method `B` is more interesting. The JIT has inlined the call from `B` to `A`, exposing the contents of `A` to `B`, such that the JIT effectively sees the body of `B` as the equivalent of `return 23 * 2;`. At that point, the JIT can do its own constant folding, and it transforms the body of `B` to simply return 46 (hex 0x2e). Constant propagation is intricately linked to constant folding and is essentially just the idea that you can substitute a constant value (typically one computed via constant folding) into further expressions, at which point they may also be able to be folded.

The JIT has long performed constant folding, but it improves further in .NET 7. One of the ways constant folding can improve is by exposing more values to be folded, which often means more inlining. dotnet/runtime#55745 helped the inliner to understand that a method call like `M(constant + constant)` (noting that those constants might be the result of some other method call) is itself passing a constant to `M`, and a constant being passed to a method call is a hint to the inliner that it should consider being more aggressive about inlining, since exposing that constant to the body of the callee can potentially significantly reduce the amount of code required to implement the callee. The JIT might have previously inlined such a method anyway, but when it comes to inlining, the JIT is all about heuristics and generating enough evidence that it's worthwhile to inline something; this contributes to that evidence. This pattern shows up, for example, in the various `FromXx` methods on `TimeSpan`. For example, `TimeSpan.FromSeconds` is implemented as:

```
public static TimeSpan FromSeconds(double value) => Interval(value, TicksPerSecond); //
TicksPerSecond is a constant
```

and, eschewing argument validation for the purposes of this example, `Interval` is:

```
private static TimeSpan Interval(double value, double scale) =>
IntervalFromDoubleTicks(value * scale);
private static TimeSpan IntervalFromDoubleTicks(double ticks) => ticks == long.MaxValue ?
TimeSpan.MaxValue : new TimeSpan((long)ticks);
```

which if everything gets inlined means `FromSeconds` is essentially:

```
public static TimeSpan FromSeconds(double value)
{
    double ticks = value * 10_000_000;
    return ticks == long.MaxValue ? TimeSpan.MaxValue : new TimeSpan((long)ticks);
}
```

and if `value` is a constant, let's say `5`, that whole thing can be constant folded (with dead code elimination on the `ticks == long.MaxValue` branch) to simply:

```
return new TimeSpan(50_000_000);
```

I'll spare you the .NET 6 assembly for this, but on .NET 7 with a benchmark like:

```
[Benchmark]
public TimeSpan FromSeconds() => TimeSpan.FromSeconds(5);
```

we now get the simple and clean:

```
; Program.FromSeconds()
       mov       eax,2FAF080
```

```
       ret
; Total bytes of code 6
```

Another change improving constant folding included [dotnet/runtime#57726](https://github.com/dotnet/runtime/pull/57726) from [@SingleAccretion](https://github.com/SingleAccretion), which unblocked constant folding in a particular scenario that sometimes manifests when doing field-by-field assignment of structs being returned from method calls. As a small example, consider this trivial property, which access the `Color.DarkOrange` property, which in turn does `new Color(KnownColor.DarkOrange)`:

```
[Benchmark]
public Color DarkOrange() => Color.DarkOrange;
```

In .NET 6, the JIT generated this:

```
; Program.DarkOrange()
       mov       eax,1
       mov       ecx,39
       xor       r8d,r8d
       mov       [rdx],r8
       mov       [rdx+8],r8
       mov       [rdx+10],cx
       mov       [rdx+12],ax
       mov       rax,rdx
       ret
; Total bytes of code 32
```

The interesting thing here is that some constants (39, which is the value of `KnownColor.DarkOrange`, and 1, which is a private `StateKnownColorValid` constant) are being loaded into registers (`mov eax, 1`, `mov ecx, 39`) and then later being stored into the relevant location for the `Color` struct being returned (`mov [rdx+12],ax` and `mov [rdx+10],cx`). In .NET 7, it now generates:

```
; Program.DarkOrange()
       xor       eax,eax
       mov       [rdx],rax
       mov       [rdx+8],rax
       mov       word ptr [rdx+10],39
       mov       word ptr [rdx+12],1
       mov       rax,rdx
       ret
; Total bytes of code 25
```

with direct assignment of these constant values into their destination locations (`mov word ptr [rdx+12],1` and `mov word ptr [rdx+10],39`). Other changes contributing to constant folding included [dotnet/runtime#58171](https://github.com/dotnet/runtime/pull/58171) from [@SingleAccretion](https://github.com/SingleAccretion) and [dotnet/runtime#57605](https://github.com/dotnet/runtime/pull/57605) from [@SingleAccretion](https://github.com/SingleAccretion).

However, a large category of improvement came from an optimization related to propagation, that of forward substitution. Consider this silly benchmark:

```
[Benchmark]
public int Compute1() => Value + Value + Value + Value + Value;

[Benchmark]
public int Compute2() => SomethingElse() + Value + Value + Value + Value + Value;

private static int Value => 16;
```

```
[MethodImpl(MethodImplOptions.NoInlining)]
private static int SomethingElse() => 42;
```

If we look at the assembly code generated for `Compute1` on .NET 6, it looks like what we'd hope for. We're adding `Value` 5 times, `Value` is trivially inlined and returns a constant value 16, and so we'd hope that the assembly code generated for `Compute1` would effectively just be returning the value 80 (hex 0x50), which is exactly what happens:

```
; Program.Compute1()
       mov       eax,50
       ret
; Total bytes of code 6
```

But `Compute2` is a bit different. The structure of the code is such that the additional call to `SomethingElse` ends up slightly perturbing something about the JIT's analysis, and .NET 6 ends up with this assembly code:

```
; Program.Compute2()
       sub       rsp,28
       call      Program.SomethingElse()
       add       eax,10
       add       eax,10
       add       eax,10
       add       eax,10
       add       eax,10
       add       rsp,28
       ret
; Total bytes of code 29
```

Rather than a single `mov eax, 50` to put the value 0x50 into the return register, we have 5 separate `add eax, 10` to build up that same 0x50 (80) value. That's… not ideal.

It turns out that many of the JIT's optimizations operate on the tree data structures created as part of parsing the IL. In some cases, optimizations can do better when they're exposed to more of the program, in other words when the tree they're operating on is larger and contains more to be analyzed. However, various operations can break up these trees into smaller, individual ones, such as with temporary variables created as part of inlining, and in doing so can inhibit these operations. Something is needed in order to effectively stitch these trees back together, and that's forward substitution. You can think of forward substitution almost like an inverse of CSE; rather than trying to find duplicate expressions and eliminate them by computing the value once and storing it into a temporary, forward substitution eliminates that temporary and effectively moves the expression tree into its use site. Obviously you don't want to do this if it would then negate CSE and result in duplicate work, but for expressions that are defined once and used once, this kind of forward propagation is valuable. dotnet/runtime#61023 added an initial limited version of forward substitution, and then dotnet/runtime#63720 added a more robust generalized implementation. Subsequently, dotnet/runtime#70587 expanded it to also cover some SIMD vectors, and then dotnet/runtime#71161 improved it further to enable substitutions into more places (in this case into call arguments). And with those, our silly benchmark now produces the following on .NET 7:

```
; Program.Compute2()
       sub       rsp,28
       call      qword ptr [7FFCB8DAF9A8]
```

```
        add       eax,50
        add       rsp,28
        ret
; Total bytes of code 18
```

Vectorization

SIMD, or Single Instruction Multiple Data, is a kind of processing in which one instruction applies to multiple pieces of data at the same time. You've got a list of numbers and you want to find the index of a particular value? You could walk the list comparing one element at a time, and that would be fine functionally. But what if in the same amount of time it takes you to read and compare one element, you could instead read and compare two elements, or four elements, or 32 elements? That's SIMD, and the art of utilizing SIMD instructions is lovingly referred to as "vectorization," where operations are applied to all of the elements in a "vector" at the same time.

.NET has long had support for vectorization in the form of `Vector<T>`, which is an easy-to-use type with first-class JIT support to enable a developer to write vectorized implementations. One of `Vector<T>`'s greatest strengths is also one of its greatest weaknesses. The type is designed to adapt to whatever width vector instructions are available in your hardware. If the machine supports 256-bit width vectors, great, that's what `Vector<T>` will target. If not, if the machine supports 128-bit width vectors, great, that's what `Vector<T>` targets. But that flexibility comes with various downsides, at least today; for example, the operations you can perform on a `Vector<T>` end up needing to be agnostic to the width of the vectors used, since the width is variable based on the hardware on which the code actually runs. And that means the operations that can be exposed on `Vector<T>` are limited, which in turn limits the kinds of operations that can be vectorized with it. Also, because it's only ever a single size in a given process, some data set sizes that fall in between 128 bits and 256 bits might not be processed as well as you'd hope. You write your `Vector<byte>`-based algorithm, and you run it on a machine with support for 256-bit vectors, which means it can process 32 bytes at a time, but then you feed it an input with 31 bytes. Had `Vector<T>` mapped to 128-bit vectors, it could have been used to improve the processing of that input, but as its vector size is larger than the input data size, the implementation ends up falling back to one that's not accelerated. There are also issues related to R2R and Native AOT, since ahead-of-time compilation needs to know in advance what instructions should be used for `Vector<T>` operations. You already saw this earlier when discussing the output of `DOTNET_JitDisasmSummary`; we saw that the `NarrowUtf16ToAscii` method was one of only a few methods that was JIT compiled in a "hello, world" console app, and that this was because it lacked R2R code due to its use of `Vector<T>`.

Starting in .NET Core 3.0, .NET gained literally thousands of new "hardware intrinsics" methods, most of which are .NET APIs that map down to one of these SIMD instructions. These intrinsics enable an expert to write an implementation tuned to a specific instruction set, and if done well, get the best possible performance, but it also requires the developer to understand each instruction set and to implement their algorithm for each instruction set that might be relevant, e.g. an AVX2 implementation if it's supported, or an SSE2 implementation if it's supported, or an ArmBase implementation if it's supported, and so on.

.NET 7 has introduced a middle ground. Previous releases saw the introduction of the `Vector128<T>` and `Vector256<T>` types, but purely as the vehicle by which data moved in and out of the hardware intrinsics, since they're all tied to specific width vectors. Now in .NET 7, exposed via dotnet/runtime#53450, dotnet/runtime#63414, dotnet/runtime#60094, and dotnet/runtime#68559, a very large set of cross-platform operations is defined over these types as well, e.g. `Vector128<T>.ExtractMostSignificantBits`, `Vector256.ConditionalSelect`, and so on. A developer who wants or needs to go beyond what the high-level `Vector<T>` offers can choose to target one or more of these two types. Typically this would amount to a developer writing one code path based on `Vector128<T>`, as that has the broadest reach and achieves a significant amount of the gains from vectorization, and then if is motivated to do so can add a second path for `Vector256<T>` in order to potentially double throughput further on platforms that have 256-bit width vectors. Think of these types and methods as a platform-abstraction layer: you code to these methods, and then the JIT translates them into the most appropriate instructions for the underlying platform. Consider this simple code as an example:

```
using System.Runtime.CompilerServices;
using System.Runtime.Intrinsics;
using System.Runtime.Intrinsics.X86;

internal class Program
{
    private static void Main()
    {
        Vector128<byte> v = Vector128.Create((byte)123);
        while (true)
        {
            WithIntrinsics(v);
            WithVector(v);
        }
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static int WithIntrinsics(Vector128<byte> v) => Sse2.MoveMask(v);

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static uint WithVector(Vector128<byte> v) => v.ExtractMostSignificantBits();
}
```

I have two functions: one that directly uses the `Sse2.MoveMask` hardware intrinsic and one that uses the new `Vector128<T>.ExtractMostSignificantBits` method. Using `DOTNET_JitDisasm=Program.*`, here's what the optimized tier-1 code for these looks like on my x64 Windows machine:

```
; Assembly listing for method Program:WithIntrinsics(Vector128`1):int
G_M000_IG01:                ;; offset=0000H
       C5F877               vzeroupper

G_M000_IG02:                ;; offset=0003H
       C5F91001             vmovupd  xmm0, xmmword ptr [rcx]
       C5F9D7C0             vpmovmskb eax, xmm0

G_M000_IG03:                ;; offset=000BH
       C3                   ret

; Total bytes of code 12

; Assembly listing for method Program:WithVector(Vector128`1):int
G_M000_IG01:                ;; offset=0000H
       C5F877               vzeroupper
```

```
G_M000_IG02:                ;; offset=0003H
       C5F91001             vmovupd  xmm0, xmmword ptr [rcx]
       C5F9D7C0             vpmovmskb eax, xmm0

G_M000_IG03:                ;; offset=000BH
       C3                   ret

; Total bytes of code 12
```

Notice anything? The code for the two methods is identical, both resulting in a `vpmovmskb` (Move Byte Mask) instruction. Yet the former code will only work on a platform that supports SSE2 whereas the latter code will work on any platform with support for 128-bit vectors, including Arm64 and WASM (and any future platforms on-boarded that also support SIMD); it'll just result in different instructions being emitted on those platforms.

To explore this a bit more, let's take a simple example and vectorize it. We'll implement a `Contains` method, where we want to search a span of bytes for a specific value and return whether it was found:

```
static bool Contains(ReadOnlySpan<byte> haystack, byte needle)
{
    for (int i = 0; i < haystack.Length; i++)
    {
        if (haystack[i] == needle)
        {
            return true;
        }
    }

    return false;
}
```

How would we vectorize this with `Vector<T>`? First things first, we need to check whether it's even supported, and fall back to our existing implementation if it's not (`Vector.IsHardwareAccelerated`). We also need to fall back if the length of the input is less than the size of a vector (`Vector<byte>.Count`).

```
static bool Contains(ReadOnlySpan<byte> haystack, byte needle)
{
    if (Vector.IsHardwareAccelerated && haystack.Length >= Vector<byte>.Count)
    {
        // ...
    }
    else
    {
        for (int i = 0; i < haystack.Length; i++)
        {
            if (haystack[i] == needle)
            {
                return true;
            }
        }
    }

    return false;
}
```

Now that we know we have enough data, we can get to coding our vectorized loop. In this loop, we'll be searching for the `needle`, which means we need a vector that contains that value for every element; the `Vector<T>`'s constructor provides that (`new Vector<byte>(needle)`). And we need to be able to slice off a vector's width of data at a time; for a bit more efficiency, I'll use pointers. We need a current iteration pointer, and we need to iterate until the point where we couldn't form another vector because we're too close to the end, and a straightforward way to do that is to get a pointer that's exactly one vector's width from the end; that way, we can just iterate until our current pointer is equal to or greater than that threshold. And finally, in our loop body, we need to compare our current vector with the target vector to see if any elements are the same (`Vector.EqualsAny`), if any is returning true, and if not bumping our current pointer to the next location. At this point we have:

```
static unsafe bool Contains(ReadOnlySpan<byte> haystack, byte needle)
{
    if (Vector.IsHardwareAccelerated && haystack.Length >= Vector<byte>.Count)
    {
        fixed (byte* haystackPtr = &MemoryMarshal.GetReference(haystack))
        {
            Vector<byte> target = new Vector<byte>(needle);
            byte* current = haystackPtr;
            byte* endMinusOneVector = haystackPtr + haystack.Length - Vector<byte>.Count;
            do
            {
                if (Vector.EqualsAny(target, *(Vector<byte>*)current))
                {
                    return true;
                }

                current += Vector<byte>.Count;
            }
            while (current < endMinusOneVector);

            // ...
        }
    }
    else
    {
        for (int i = 0; i < haystack.Length; i++)
        {
            if (haystack[i] == needle)
            {
                return true;
            }
        }
    }

    return false;
}
```

And we're almost done. The last issue to handle is we may still have a few elements at the end we haven't searched. There are a couple of ways we could handle that. One would be to just continue with our fall back implementation and process each of the remaining elements one at a time. Another would be to employ a trick that's common when vectorizing idempotent operations. Our operation isn't mutating anything, which means it doesn't matter if we compare the same element multiple times, which means we can just do one final vector compare for the last vector in the search space;

that might or might not overlap with elements we've already looked at, but it won't hurt anything if it does. And with that, our implementation is complete:

```
static unsafe bool Contains(ReadOnlySpan<byte> haystack, byte needle)
{
    if (Vector.IsHardwareAccelerated && haystack.Length >= Vector<byte>.Count)
    {
        fixed (byte* haystackPtr = &MemoryMarshal.GetReference(haystack))
        {
            Vector<byte> target = new Vector<byte>(needle);
            byte* current = haystackPtr;
            byte* endMinusOneVector = haystackPtr + haystack.Length - Vector<byte>.Count;
            do
            {
                if (Vector.EqualsAny(target, *(Vector<byte>*)current))
                {
                    return true;
                }

                current += Vector<byte>.Count;
            }
            while (current < endMinusOneVector);

            if (Vector.EqualsAny(target, *(Vector<byte>*)endMinusOneVector))
            {
                return true;
            }
        }
    }
    else
    {
        for (int i = 0; i < haystack.Length; i++)
        {
            if (haystack[i] == needle)
            {
                return true;
            }
        }
    }

    return false;
}
```

Congratulations, we've vectorized this operation, and fairly decently at that. We can throw this into benchmarkdotnet and see really nice speedups:

```
private byte[] _data = Enumerable.Repeat((byte)123, 999).Append((byte)42).ToArray();

[Benchmark(Baseline = true)]
[Arguments((byte)42)]
public bool Find(byte value) => Contains(_data, value); // just the fallback path in its
own method

[Benchmark]
[Arguments((byte)42)]
public bool FindVectorized(byte value) => Contains_Vectorized(_data, value); // the
implementation we just wrote
```

Method	Mean	Ratio
Find	484.05 ns	1.00
FindVectorized	20.21 ns	0.04

A 24x speedup! Woo hoo, victory, all your performance are belong to us!

You deploy this in your service, and you see `Contains` being called on your hot path, but you don't see the improvements you were expecting. You dig in a little more, and you discover that while you tested this with an input array with 1000 elements, typical inputs had more like 30 elements. What happens if we change our benchmark to have just 30 elements? That's not long enough to form a vector, so we fall back to the one-at-a-time path, and we don't get any speedups at all.

One thing we can now do is switch from using `Vector<T>` to `Vector128<T>`. That will then lower the threshold from 32 bytes to 16 bytes, such that inputs in that range will still have some amount of vectorization applied. As these `Vector128<T>` and `Vector256<T>` types have been designed very recently, they also utilize all the cool new toys, and thus we can use `ref`s instead of pointers. Other than that, we can keep the shape of our implementation almost the same, substituting `Vector128` where we were using `Vector`, and using some methods on `Unsafe` to manipulate our `ref`s instead of pointer arithmetic on the span we fixed.

```
static unsafe bool Contains(ReadOnlySpan<byte> haystack, byte needle)
{
    if (Vector128.IsHardwareAccelerated && haystack.Length >= Vector128<byte>.Count)
    {
        ref byte current = ref MemoryMarshal.GetReference(haystack);

        Vector128<byte> target = Vector128.Create(needle);
        ref byte endMinusOneVector = ref Unsafe.Add(ref current, haystack.Length -
Vector128<byte>.Count);
        do
        {
            if (Vector128.EqualsAny(target, Vector128.LoadUnsafe(ref current)))
            {
                return true;
            }

            current = ref Unsafe.Add(ref current, Vector128<byte>.Count);
        }
        while (Unsafe.IsAddressLessThan(ref current, ref endMinusOneVector));

        if (Vector128.EqualsAny(target, Vector128.LoadUnsafe(ref endMinusOneVector)))
        {
            return true;
        }
    }
    else
    {
        for (int i = 0; i < haystack.Length; i++)
        {
            if (haystack[i] == needle)
            {
                return true;
            }
        }
    }
```

```
    return false;
}
```

With that in hand, we can now try it on our smaller 30 element data set:

```
private byte[] _data = Enumerable.Repeat((byte)123, 29).Append((byte)42).ToArray();

[Benchmark(Baseline = true)]
[Arguments((byte)42)]
public bool Find(byte value) => Contains(_data, value);

[Benchmark]
[Arguments((byte)42)]
public bool FindVectorized(byte value) => Contains_Vectorized(_data, value);
```

Method	Mean	Ratio
Find	15.388 ns	1.00
FindVectorized	1.747 ns	0.11

Woo hoo, victory, all your performance are belong to us… again!

What about on the larger data set again? Previously with `Vector<T>` we had a 24x speedup, but now:

Method	Mean	Ratio
Find	484.25 ns	1.00
FindVectorized	32.92 ns	0.07

… closer to 15x. Nothing to sneeze at, but it's not the 24x we previously saw. What if we want to have our cake and eat it, too? Let's also add a `Vector256<T>` path. To do that, we literally copy/paste our `Vector128<T>` code, search/replace all references to `Vector128` in the copied code with `Vector256`, and just put it into an additional condition that uses the `Vector256<T>` path if it's supported and there are enough elements to utilize it.

```
static unsafe bool Contains(ReadOnlySpan<byte> haystack, byte needle)
{
    if (Vector128.IsHardwareAccelerated && haystack.Length >= Vector128<byte>.Count)
    {
        ref byte current = ref MemoryMarshal.GetReference(haystack);

        if (Vector256.IsHardwareAccelerated && haystack.Length >= Vector256<byte>.Count)
        {
            Vector256<byte> target = Vector256.Create(needle);
            ref byte endMinusOneVector = ref Unsafe.Add(ref current, haystack.Length -
Vector256<byte>.Count);
            do
            {
                if (Vector256.EqualsAny(target, Vector256.LoadUnsafe(ref current)))
                {
                    return true;
                }

                current = ref Unsafe.Add(ref current, Vector256<byte>.Count);
            }
            while (Unsafe.IsAddressLessThan(ref current, ref endMinusOneVector));
```

```
            if (Vector256.EqualsAny(target, Vector256.LoadUnsafe(ref endMinusOneVector)))
            {
                return true;
            }
        }
        else
        {
            Vector128<byte> target = Vector128.Create(needle);
            ref byte endMinusOneVector = ref Unsafe.Add(ref current, haystack.Length -
Vector128<byte>.Count);
            do
            {
                if (Vector128.EqualsAny(target, Vector128.LoadUnsafe(ref current)))
                {
                    return true;
                }

                current = ref Unsafe.Add(ref current, Vector128<byte>.Count);
            }
            while (Unsafe.IsAddressLessThan(ref current, ref endMinusOneVector));

            if (Vector128.EqualsAny(target, Vector128.LoadUnsafe(ref endMinusOneVector)))
            {
                return true;
            }
        }
    }
    else
    {
        for (int i = 0; i < haystack.Length; i++)
        {
            if (haystack[i] == needle)
            {
                return true;
            }
        }
    }

    return false;
}
```

And, boom, we're back:

Method	Mean	Ratio
Find	484.53 ns	1.00
FindVectorized	20.08 ns	0.04

We now have an implementation that is vectorized on any platform with either 128-bit or 256-bit vector instructions (x86, x64, Arm64, WASM, etc.), that can use either based on the input length, and that can be included in an R2R image if that's of interest.

There are many factors that impact which path you go down, and I expect we'll have guidance forthcoming to help navigate all the factors and approaches. But the capabilities are all there, and whether you choose to use `Vector<T>`, `Vector128<T>` and/or `Vector256<T>`, or the hardware intrinsics directly, there are some amazing performance opportunities ready for the taking.

I already mentioned several PRs that exposed the new cross-platform vector support, but that only scratches the surface of the work done to actually enable these operations and to enable them to produce high-quality code. As just one example of a category of such work, a set of changes went in to help ensure that zero vector constants are handled well, such as [dotnet/runtime#63821] that "morphed" (changed) `Vector128/256<T>.Create(default)` into `Vector128/256<T>.Zero`, which then enables subsequent optimizations to focus only on `Zero`; [dotnet/runtime#65028] that enabled constant propagation of `Vector128/256<T>.Zero`; [dotnet/runtime#68874] and [dotnet/runtime#70171] that add first-class knowledge of vector constants to the JIT's intermediate representation; and [dotnet/runtime#62933], [dotnet/runtime#65632], [dotnet/runtime#55875], [dotnet/runtime#67502], and [dotnet/runtime#64783] that all improve the code quality of instructions generated for zero vector comparisons.

Inlining

Inlining is one of the most important optimizations the JIT can do. The concept is simple: instead of making a call to some method, take the code from that method and bake it into the call site. This has the obvious advantage of avoiding the overhead of a method call, but except for really small methods on really hot paths, that's often on the smaller side of the wins inlining brings. The bigger wins are due to the callee's code being exposed to the caller's code, and vice versa. So, for example, if the caller is passing a constant as an argument to the callee, if the method isn't inlined, the compilation of the callee has no knowledge of that constant, but if the callee is inlined, all of the code in the callee is then aware of its argument being a constant value, and can do all of the optimizations possible with such a constant, like dead code elimination, branch elimination, constant folding and propagation, and so on. Of course, if it were all rainbows and unicorns, everything possible to be inlined would be inlined, and that's obviously not happening. Inlining brings with it the cost of potentially increased binary size. If the code being inlined would result in the same amount or less assembly code in the caller than it takes to call the callee (and if the JIT can quickly determine that), then inlining is a no-brainer. But if the code being inlined would increase the size of the callee non-trivially, now the JIT needs to weigh that increase in code size against the throughput benefits that could come from it. That code size increase can itself result in throughput regressions, due to increasing the number of distinct instructions to be executed and thereby putting more pressure on the instruction cache. As with any cache, the more times you need to read from memory to populate it, the less effective the cache will be. If you have a function that gets inlined into 100 different call sites, every one of those call sites' copies of the callee's instructions are unique, and calling each of those 100 functions could end up thrashing the instruction cache; in contrast, if all of those 100 functions "shared" the same instructions by simply calling the single instance of the callee, it's likely the instruction cache would be much more effective and lead to fewer trips to memory.

All that is to say, inlining is *really* important, it's important that the "right" things be inlined and that it not overinline, and as such every release of .NET in recent memory has seen nice improvements around inlining. .NET 7 is no exception.

One really interesting improvement around inlining is [dotnet/runtime#64521], and it might be surprising. Consider the `Boolean.ToString` method; here's its full implementation:

```
public override string ToString()
{
```

```
    if (!m_value) return "False";
    return "True";
}
```

Pretty simple, right? You'd expect something this trivial to be inlined. Alas, on .NET 6, this benchmark:

```
private bool _value = true;

[Benchmark]
public int BoolStringLength() => _value.ToString().Length;
```

produces this assembly code:

```
; Program.BoolStringLength()
       sub       rsp,28
       cmp       [rcx],ecx
       add       rcx,8
       call      System.Boolean.ToString()
       mov       eax,[rax+8]
       add       rsp,28
       ret
; Total bytes of code 23
```

Note the `call System.Boolean.ToString()`. The reason for this is, historically, the JIT has been unable to inline methods across assembly boundaries if those methods contain string literals (like the `"False"` and `"True"` in that `Boolean.ToString` implementation). This restriction had to do with string interning and the possibility that such inlining could lead to visible behavioral differences. Those concerns are no longer valid, and so this PR removes the restriction. As a result, that same benchmark on .NET 7 now produces this:

```
; Program.BoolStringLength()
       cmp       byte ptr [rcx+8],0
       je        short M00_L01
       mov       rax,1DB54800D20
       mov       rax,[rax]
M00_L00:
       mov       eax,[rax+8]
       ret
M00_L01:
       mov       rax,1DB54800D18
       mov       rax,[rax]
       jmp       short M00_L00
; Total bytes of code 38
```

No more `call System.Boolean.ToString()`.

[dotnet/runtime#61408](https://github.com/dotnet/runtime/pull/61408) made two changes related to inlining. First, it taught the inliner how to better see the what methods were being called in an inlining candidate, and in particular when tiered compilation is disabled or when a method would bypass tier-0 (such as a method with loops before OSR existed or with OSR disabled); by understanding what methods are being called, it can better understand the cost of the method, e.g. if those method calls are actually hardware intrinsics with a very low cost. Second, it enabled CSE in more cases with SIMD vectors.

[dotnet/runtime#71778](https://github.com/dotnet/runtime/pull/71778) also impacted inlining, and in particular in situations where a `typeof()` could be propagated to the callee (e.g. via a method argument). In previous releases of .NET, various members on `Type` like `IsValueType` were turned into JIT intrinsics, such that the JIT could substitute a constant value for calls where it could compute the answer at compile time. For example, this:

```
[Benchmark]
public bool IsValueType() => IsValueType<int>();

private static bool IsValueType<T>() => typeof(T).IsValueType;
```

results in this assembly code on .NET 6:

```
; Program.IsValueType()
       mov       eax,1
       ret
; Total bytes of code 6
```

However, change the benchmark slightly:

```
[Benchmark]
public bool IsValueType() => IsValueType(typeof(int));

private static bool IsValueType(Type t) => t.IsValueType;
```

and it's no longer as simple:

```
; Program.IsValueType()
       sub       rsp,28
       mov       rcx,offset MT_System.Int32
       call      CORINFO_HELP_TYPEHANDLE_TO_RUNTIMETYPE
       mov       rcx,rax
       mov       rax,[7FFCA47C9560]
       cmp       [rcx],ecx
       add       rsp,28
       jmp       rax
; Total bytes of code 38
```

Effectively, as part of inlining the JIT loses the notion that the argument is a constant and fails to propagate it. This PR fixes that, such that on .NET 7, we now get what we expect:

```
; Program.IsValueType()
       mov       eax,1
       ret
; Total bytes of code 6
```

Arm64

A huge amount of effort in .NET 7 went into making code gen for Arm64 as good or better than its x64 counterpart. I've already discussed a bunch of PRs that are relevant regardless of architecture, and others that are specific to Arm, but there are plenty more. To rattle off some of them:

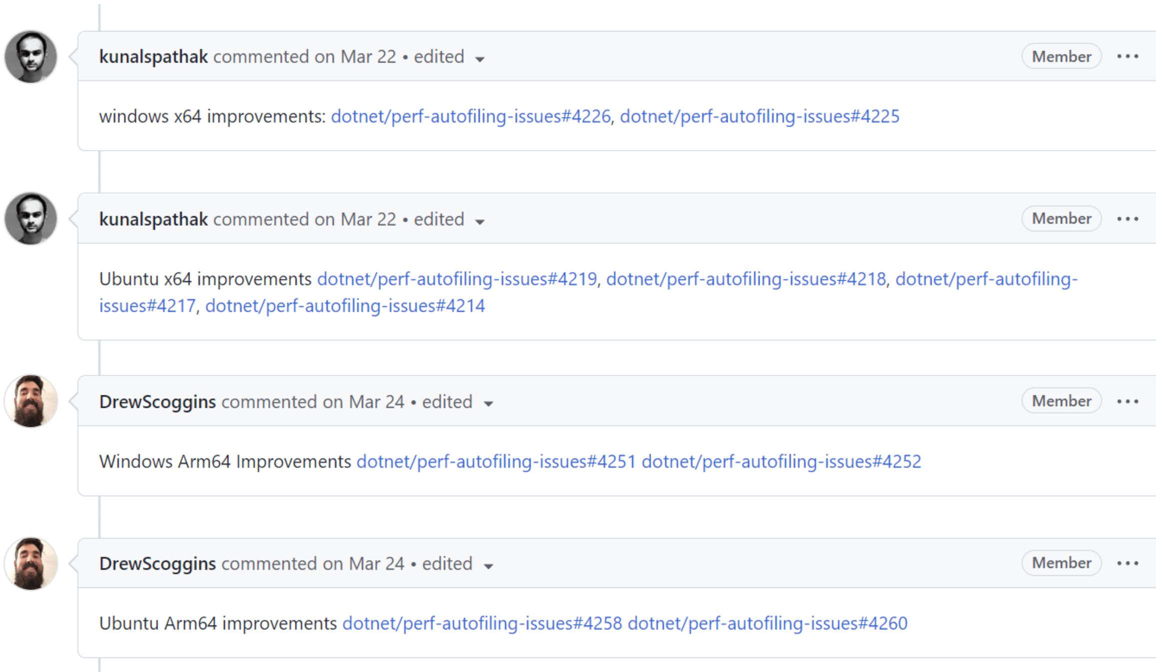
- **Addressing modes**. "Addressing mode" is the term used to refer to how the operand of instructions are specified. It could be the actual value, it could be the address from where a value should be loaded, it could be the register containing the value, and so on. Arm supports a "scaled" addressing mode, typically used for indexing into an array, where the size of each element is supplied and the instruction "scales" the provided offset by the specified scale. [dotnet/runtime#60808](https://github.com/dotnet/runtime/pull/60808) enables the JIT to utilize this addressing mode. More generally, [dotnet/runtime#70749](https://github.com/dotnet/runtime/pull/70749) enables the JIT to use addressing modes when accessing elements of managed arrays. dotnet/runtime#66902 improves the use of addressing modes when the element type is `byte`. dotnet/runtime#65468 improves addressing modes used for floating point. And dotnet/runtime#67490 implements addressing modes for SIMD vectors, specifically for loads with unscaled indices.

- **Better instruction selection**. Various techniques go into ensuring that the best instructions are selected to represent input code. dotnet/runtime#61037 teaches the JIT how to recognize the pattern `(a * b) + c` with integers and fold that into a single `madd` or `msub` instruction, while dotnet/runtime#66621 does the same for `a - (b * c)` and `msub`. dotnet/runtime#61045 enables the JIT to recognize certain constant bit shift operations (either explicit in the code or implicit to various forms of managed array access) and emit `sbfiz`/`ubfiz` instructions. dotnet/runtime#70599, dotnet/runtime#66407, and dotnet/runtime#65535 all handle various forms of optimizing `a % b`. dotnet/runtime#61847 from [@SeanWoo](https://github.com/SeanWoo) removes an unnecessary `movi` emitted as part of setting a dereferenced pointer to a constant value. dotnet/runtime#57926 from [@SingleAccretion](https://github.com/SingleAccretion) enables computing a 64-bit result as the multiplication of two 32-bit integers to be done with `smull`/`umull`. And dotnet/runtime#61549 folds adds with sign extension or zero extension into a single `add` instruction with `uxtw`/`sxtw`/`lsl`, while dotnet/runtime#62630 drops redundant zero extensions after a `ldr` instruction.
- **Vectorization**. dotnet/runtime#64864 adds new `AdvSimd.LoadPairVector64`/`AdvSimd.LoadPairVector128` hardware intrinsics.
- **Zeroing**. Lots of operations require state to be set to zero, such as initializing all reference locals in a method to zero as part of the method's prologue (so that the GC doesn't see and try to follow garbage references). While such functionality was previously vectorized, dotnet/runtime#63422 enables this to be implemented using 128-bit width vector instructions on Arm. And dotnet/runtime#64481 changes the instruction sequences used for zeroing in order to avoid unnecessary zeroing, free up additional registers, and enable the CPU to recognize various instruction sequences and better optimize.
- **Memory Model**. dotnet/runtime#62895 enables store barriers to be used wherever possible instead of full barriers, and uses one-way barriers for `volatile` variables. dotnet/runtime#67384 enables volatile reads/writes to be implemented with the `ldapr` instruction, while dotnet/runtime#64354 uses a cheaper instruction sequence to handle volatile indirections. There's dotnet/runtime#70600, which enables LSE Atomics to be used for `Interlocked` operations; dotnet/runtime#71512, which enables using the `atomics` instruction on Unix machines; and dotnet/runtime#70921, which enables the same but on Windows.

JIT helpers

While logically part of the runtime, the JIT is actually isolated from the rest of the runtime, only interacting with it through an interface that enables communication between the JIT and the rest of the VM (Virtual Machine). There's a large amount of VM functionality then that the JIT relies on for good performance.

dotnet/runtime#65738 rewrote various "stubs" to be more efficient. Stubs are tiny bits of code that serve to perform some check and then redirect execution somewhere else. For example, when an interface dispatch call site is expected to only ever be used with a single implementation of that interface, the JIT might employ a "dispatch stub" that compares the type of the object against the single one it's cached, and if they're equal simply jumps to the right target. You know you're in the corest of the core areas of the runtime when a PR contains lots of assembly code for every architecture the runtime targets. And it paid off; there's a virtual group of folks from around .NET that review performance improvements and regressions in our automated performance test suites, and attribute these back to the PRs likely to be the cause (this is mostly automated but requires some human oversight). It's always nice then when a few days after a PR is merged and performance information has stabilized that you see a rash of comments like there were on this PR:


kunalspathak commented on Mar 22 • edited Member

windows x64 improvements: dotnet/perf-autofiling-issues#4226, dotnet/perf-autofiling-issues#4225

kunalspathak commented on Mar 22 • edited Member

Ubuntu x64 improvements dotnet/perf-autofiling-issues#4219, dotnet/perf-autofiling-issues#4218, dotnet/perf-autofiling-issues#4217, dotnet/perf-autofiling-issues#4214

DrewScoggins commented on Mar 24 • edited Member

Windows Arm64 Improvements dotnet/perf-autofiling-issues#4251 dotnet/perf-autofiling-issues#4252

DrewScoggins commented on Mar 24 • edited Member

Ubuntu Arm64 improvements dotnet/perf-autofiling-issues#4258 dotnet/perf-autofiling-issues#4260

For anyone familiar with generics and interested in performance, you may have heard the refrain that generic virtual methods are relatively expensive. They are, comparatively. For example on .NET 6, this code:

```
private Example _example = new Example();

[Benchmark(Baseline = true)] public void GenericNonVirtual() =>
_example.GenericNonVirtual<Example>();
[Benchmark] public void GenericVirtual() => _example.GenericVirtual<Example>();

class Example
{
    [MethodImpl(MethodImplOptions.NoInlining)]
    public void GenericNonVirtual<T>() { }

    [MethodImpl(MethodImplOptions.NoInlining)]
    public virtual void GenericVirtual<T>() { }
}
```

results in:

Method	Mean	Ratio
GenericNonVirtual	0.4866 ns	1.00
GenericVirtual	6.4552 ns	13.28

dotnet/runtime#65926 eases the pain a tad. Some of the cost comes from looking up some cached information in a hash table in the runtime, and as is the case with many map implementations, this one involves computing a hash code and using a mod operation to map to the right bucket. Other hash table implementations around dotnet/runtime, including `Dictionary<,>`, `HashSet<,>`, and `ConcurrentDictionary<,>` previously switched to a "fastmod" implementation; this PR does the same for this `EEHashtable`, which is used as part of the `CORINFO_GENERIC_HANDLE` JIT helper function employed:

Method	Runtime	Mean	Ratio
GenericVirtual	.NET 6.0	6.475 ns	1.00
GenericVirtual	.NET 7.0	6.119 ns	0.95

Not enough of an improvement for us to start recommending people use them, but a 5% improvement takes a bit of the edge off the sting.

Grab Bag

It's near impossible to cover every performance change that goes into the JIT, and I'm not going to try. But there were so many more PRs, I couldn't just leave them all unsung, so here's a few more quickies:

- dotnet/runtime#58196 from [@benjamin-hodgson](https://github.com/benjamin-hodgson). Given an expression like `(byte)x | (byte)y`, that can be morphed into `(byte)(x | y)`, which can optimize away some `movs`.

```
private int _x, _y;

[Benchmark]
public int Test() => (byte)_x | (byte)_y;
```

```
; *** .NET 6 ***
; Program.Test(Int32, Int32)
       movzx     eax,dl
       movzx     edx,r8b
       or        eax,edx
       ret
; Total bytes of code 10

; *** .NET 7 ***
; Program.Test(Int32, Int32)
       or        edx,r8d
       movzx     eax,dl
       ret
; Total bytes of code 7
```

- [dotnet/runtime#67182](https://github.com/dotnet/runtime/pull/67182). On a machine with support for BMI2, 64-bit shifts can be performed with the `shlx`, `sarx`, and `shrx` instructions.

```
[Benchmark]
[Arguments(123, 1)]
public ulong Shift(ulong x, int y) => x << y;
```

```
; *** .NET 6 ***
; Program.Shift(UInt64, Int32)
       mov       ecx,r8d
       mov       rax,rdx
       shl       rax,cl
       ret
; Total bytes of code 10

; *** .NET 7 ***
; Program.Shift(UInt64, Int32)
       shlx      rax,rdx,r8
       ret
; Total bytes of code 6
```

- [dotnet/runtime#69003](https://github.com/dotnet/runtime/pull/69003) from [@SkiFoD](https://github.com/SkiFoD). The pattern `~x + 1` can be changed into a two's-complement negation.

```
[Benchmark]
[Arguments(42)]
public int Neg(int i) => ~i + 1;
```

```
; *** .NET 6 ***
; Program.Neg(Int32)
       mov       eax,edx
       not       eax
       inc       eax
       ret
; Total bytes of code 7

; *** .NET 7 ***
; Program.Neg(Int32)
       mov       eax,edx
       neg       eax
       ret
; Total bytes of code 5
```

- [dotnet/runtime#61412](https://github.com/dotnet/runtime/pull/61412) from [@SkiFoD](https://github.com/SkiFoD). An expression `X & 1 == 1` to test whether the bottom bit of a number is set can changed to the cheaper `X & 1` (which isn't actually expressible without a following `!= 0` in C#).

```
[Benchmark]
[Arguments(42)]
public bool BitSet(int x) => (x & 1) == 1;
```

```
; *** .NET 6 ***
; Program.BitSet(Int32)
       test      dl,1
       setne     al
```

```
        movzx     eax,al
        ret
; Total bytes of code 10

; *** .NET 7 ***
; Program.BitSet(Int32)
        mov       eax,edx
        and       eax,1
        ret
; Total bytes of code 6
```

- dotnet/runtime#63545 from [@Wraith2](https://github.com/Wraith2). The expression `x & (x - 1)` can be lowered to the `blsr` instruction.

```
[Benchmark]
[Arguments(42)]
public int ResetLowestSetBit(int x) => x & (x - 1);
```

```
; *** .NET 6 ***
; Program.ResetLowestSetBit(Int32)
        lea       eax,[rdx+0FFFF]
        and       eax,edx
        ret
; Total bytes of code 6

; *** .NET 7 ***
; Program.ResetLowestSetBit(Int32)
        blsr      eax,edx
        ret
; Total bytes of code 6
```

- dotnet/runtime#62394. `/` and `%` by a vector's `.Count` wasn't recognizing that `Count` can be unsigned, but doing so leads to better code gen.

```
[Benchmark]
[Arguments(42u)]
public long DivideByVectorCount(uint i) => i / Vector<byte>.Count;
```

```
; *** .NET 6 ***
; Program.DivideByVectorCount(UInt32)
        mov       eax,edx
        mov       rdx,rax
        sar       rdx,3F
        and       rdx,1F
        add       rax,rdx
        sar       rax,5
        ret
; Total bytes of code 21

; *** .NET 7 ***
; Program.DivideByVectorCount(UInt32)
        mov       eax,edx
        shr       rax,5
        ret
; Total bytes of code 7
```

- dotnet/runtime#60787. Loop alignment in .NET 6 provides a very nice exploration of why and how the JIT handles loop alignment. This PR extends that further by trying to "hide" an emitted `align` instruction behind an unconditional `jmp` that might already exist, in order to minimize the impact of the processor having to fetch and decode `nop`s.

GC

"Regions" is a feature of the garbage collector (GC) that's been in the works for multiple years. It's enabled by default in 64-bit processes in .NET 7 as of [dotnet/runtime#64688](), but as with other multi-year features, a multitude of PRs went into making it a reality. At a 30,000 foot level, "regions" replaces the current "segments" approach to managing memory on the GC heap; rather than having a few gigantic segments of memory (e.g. each 1GB), often associated 1:1 with a generation, the GC instead maintains many, many smaller regions (e.g. each 4MB) as their own entity. This enables the GC to be more agile with regards to operations like repurposing regions of memory from one generation to another. For more information on regions, the blog post [Put a DPAD on that GC!]() from the primary developer on the GC is still the best resource.

Native AOT

To many people, the word "performance" in the context of software is about throughput. How fast does something execute? How much data per second can it process? How many requests per second can it process? And so on. But there are many other facets to performance. How much memory does it consume? How fast does it start up and get to the point of doing something useful? How much space does it consume on disk? How long would it take to download? And then there are related concerns. In order to achieve these goals, what dependencies are required? What kinds of operations does it need to perform to achieve these goals, and are all of those operations permitted in the target environment? If any of this paragraph resonates with you, you are the target audience for the Native AOT support now shipping in .NET 7.

.NET has long had support for AOT code generation. For example, .NET Framework had it in the form of `ngen`, and .NET Core has it in the form of `crossgen`. Both of those solutions involve a standard .NET executable that has some of its IL already compiled to assembly code, but not all methods will have assembly code generated for them, various things can invalidate the assembly code that was generated, external .NET assemblies without any native assembly code can be loaded, and so on, and in all of those cases, the runtime continues to utilize a JIT compiler. Native AOT is different. It's an evolution of CoreRT, which itself was an evolution of .NET Native, and it's entirely free of a JIT. The binary that results from publishing a build is a completely standalone executable in the target platform's platform-specific file format (e.g. COFF on Windows, ELF on Linux, Mach-O on macOS) with no external dependencies other than ones standard to that platform (e.g. libc). And it's entirely native: no IL in sight, no JIT, no nothing. All required code is compiled and/or linked in to the executable, including the same GC that's used with standard .NET apps and services, and a minimal runtime that provides services around threading and the like. All of that brings great benefits: super fast startup time, small and entirely-self contained deployment, and ability to run in places JIT compilers aren't allowed (e.g. because memory pages that were writable can't then be executable). It also brings limitations: no JIT means no dynamic loading of arbitrary assemblies (e.g. `Assembly.LoadFile`) and no reflection emit (e.g. `DynamicMethod`), everything compiled and linked in to the app means the more functionality that's used (or might be used) the larger is your deployment, etc. Even with those limitations, for a certain class of application, Native AOT is an incredibly exciting and welcome addition to .NET 7.

Too many PRs to mention have gone into bringing up the Native AOT stack, in part because it's been in the works for years (as part of the archived dotnet/corert project and then as part of dotnet/runtimelab/feature/NativeAOT) and in part because there have been over a hundred PRs just in dotnet/runtime that have gone into bringing Native AOT up to a shippable state since the code was originally brought over from dotnet/runtimelab in dotnet/runtime#62563 and dotnet/runtime#62563. Between that and there not being a previous version to compare its performance to, instead of focusing PR by PR on improvements, let's just look at how to use it and the benefits it brings.

Today, Native AOT is focused on console applications, so let's create a console app:

```
dotnet new console -o nativeaotexample
```

We now have our `nativeaotexample` directory containing a `nativeaotexample.csproj` and a "hello, world" `Program.cs`. To enable publishing the application with Native AOT, edit the .csproj to include this in the existing `<PropertyGroup>...</PropertyGroup>`.

```
<PublishAot>true</PublishAot>
```

And then... actually, that's it. Our app is now fully configured to be able to target Native AOT. All that's left is to publish. As I'm currently writing this on my Windows x64 machine, I'll target that:

```
dotnet publish -r win-x64 -c Release
```

I now have my generated executable in the output publish directory:

```
    Directory: C:\nativeaotexample\bin\Release\net7.0\win-x64\publish

Mode                 LastWriteTime         Length Name
-a---          8/27/2022  6:19 PM        2061824 nativeaotexample.exe
-a---          8/27/2022  6:19 PM       14290944 nativeaotexample.pdb
```

so 2M instead of 3.5MB. Of course, for that significant reduction I've given up some things:

- Setting `InvariantGlobalization` to true means I'm now not respecting culture information and am instead using a set of invariant data for most globalization operations.
- Setting `UseSystemResourceKeys` to true means nice exception messages are stripped away.
- Setting `IlcGenerateStackTraceData` to false means I'm going to get fairly poor stack traces should I need to debug an exception.
- Setting `DebuggerSupport` to false… good luck debugging things.
- … you get the idea.

One of the potentially mind-boggling aspects of Native AOT for a developer used to .NET is that, as it says on the tin, it really is native. After publishing the app, there is no IL involved, and there's no JIT that could even process it. This makes some of the other investments in .NET 7 all the more valuable, for example everywhere investments are happening in source generators. Code that previously relied on reflection emit for good performance will need another scheme. We can see that, for example, with `Regex`. Historically for optimal throughput with `Regex`, it's been recommended to use `RegexOptions.Compiled`, which uses reflection emit at run-time to generate an optimized implementation of the specified pattern. But if you look at the implementation of the `Regex` constructor, you'll find this nugget:

```
if (RuntimeFeature.IsDynamicCodeCompiled)
{
    factory = Compile(pattern, tree, options, matchTimeout != InfiniteMatchTimeout);
}
```

With the JIT, `IsDynamicCodeCompiled` is true. But with Native AOT, it's false. Thus, with Native AOT and `Regex`, there's no difference between specifying `RegexOptions.Compiled` and not, and another mechanism is required to get the throughput benefits promised by `RegexOptions.Compiled`. Enter `[GeneratedRegex(...)]`, which, along with the new regex source generator shipping in the .NET 7

SDK, emits C# code into the assembly using it. That C# code takes the place of the reflection emit that would have happened at run-time, and is thus able to work successfully with Native AOT.

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;

private Regex _interpreter = new Regex(@"^.*elementary.*$", RegexOptions.Multiline);

private Regex _compiled = new Regex(@"^.*elementary.*$", RegexOptions.Compiled |
RegexOptions.Multiline);

[GeneratedRegex(@"^.*elementary.*$", RegexOptions.Multiline)]
private partial Regex SG();

[Benchmark(Baseline = true)] public int Interpreter() => _interpreter.Count(s_haystack);

[Benchmark] public int Compiled() => _compiled.Count(s_haystack);

[Benchmark] public int SourceGenerator() => SG().Count(s_haystack);
```

Method	Mean	Ratio
Interpreter	9,036.7 us	1.00
Compiled	9,064.8 us	1.00
SourceGenerator	426.1 us	0.05

So, yes, there are some constraints associated with Native AOT, but there are also solutions for working with those constraints. And further, those constraints can actually bring further benefits. Consider dotnet/runtime#64497. Remember how we talked about "guarded devirtualization" in dynamic PGO, where via instrumentation the JIT can determine the most likely type to be used at a given call site and special-case it? With Native AOT, the entirety of the program is known at compile time, with no support for `Assembly.LoadFrom` or the like. That means at compile time, the compiler can do whole-program analysis to determine what types implement what interfaces. If a given interface only has a single type that implements it, then every call site through that interface can be unconditionally devirtualized, without any type-check guards.

This is a really exciting space, one we expect to see flourish in coming releases.

Mono

Up until now I've referred to "the JIT," "the GC," and "the runtime," but in reality there are actually multiple runtimes in .NET. I've been talking about "coreclr," which is the runtime that's recommended for use on Linux, macOS, and Windows. However, there's also "mono," which powers Blazor wasm applications, Android apps, and iOS apps. It's also seen significant improvements in .NET 7.

Just as with coreclr (which can JIT compile, AOT compile partially with JIT fallback, and fully Native AOT compile), mono has multiple ways of actually executing code. One of those ways is an interpreter, which enables mono to execute .NET code in environments that don't permit JIT'ing and without requiring ahead-of-time compilation or incurring any limitations it may bring. Interestingly, though, the interpreter is itself almost a full-fledged compiler, parsing the IL, generating its own intermediate representation (IR) for it, and doing one or more optimization passes over that IR; it's just that at the end of the pipeline when a compiler would normally emit code, the interpreter instead saves off that data for it to interpret when the time comes to run. As such, the interpreter has a very similar conundrum to the one we discussed with coreclr's JIT: the time it takes to optimize vs the desire to start up quickly. And in .NET 7, the interpreter employs a similar solution: tiered compilation. dotnet/runtime#68823 adds the ability for the interpreter to initially compile with minimal optimization of that IR, and then once a certain threshold of call counts has been hit, then take the time to do as much optimization on the IR as possible for all future invocations of that method. This yields the same benefits as it does for coreclr: improved startup time while also having efficient sustained throughput. When this merged, we saw improvements in Blazor wasm app startup time improve by 10-20%. Here's one example from an app being tracked in our benchmarking system:

The interpreter isn't just used for entire apps, though. Just as how coreclr can use the JIT when an R2R image doesn't contain code for a method, mono can use the interpreter when there's no AOT code for a method. Once such case that occurred on mono was with generic delegate invocation, where the presence of a generic delegate being invoked would trigger falling back to the interpreter; for .NET 7, that gap was addressed with [dotnet/runtime#70653](). A more impactful case, however, is [dotnet/runtime#64867](). Previously, any methods with `catch` or `filter` exception handling clauses couldn't be AOT compiled and would fall back to being interpreted. With this PR, the method is now able to be AOT compiled, and it only falls back to using the interpreter when an exception actually occurs, switching over to the interpreter for the remainder of that method call's execution. Since many methods contain such clauses, this can make a big difference in throughput and CPU consumption. In the same vein, [dotnet/runtime#63065]() enabled methods with `finally` exception handling clauses to be AOT compiled; just the `finally` block gets interpreted rather than the entire method being interpreted.

Beyond such backend improvements, another class of improvement came from further unification between coreclr and mono. Years ago, coreclr and mono had their own entire library stack built on top of them. Over time, as .NET was open sourced, portions of mono's stack got replaced by shared components, bit by bit. Fast forward to today, all of the core .NET libraries above `System.Private.CoreLib` are the same regardless of which runtime is being employed. In fact, the source for `CoreLib` itself is almost entirely shared, with ~95% of the source files being compiled into the `CoreLib` that's built for each runtime, and just a few percent of the source specialized for each (these statements means that the vast majority of the performance improvements discussed in the rest of this post apply equally whether running on mono and coreclr). Even so, every release now we try to chip away at that few remaining percent, for reasons of maintainability, but also because the source used for coreclr's `CoreLib` has generally had more attention paid to it from a performance perspective. [dotnet/runtime#71325](), for example, moves mono's array and span sorting generic sorting utility class over to the more efficient implementation used by coreclr.

One of the biggest categories of improvements, however, is in vectorization. This comes in two pieces. First, `Vector<T>` and `Vector128<T>` are now fully accelerated on both x64 and Arm64, thanks to PRs like [dotnet/runtime#64961](), [dotnet/runtime#65086](), [dotnet/runtime#65128](), [dotnet/runtime#66317](), dotnet/runtime#66391, dotnet/runtime#66409, dotnet/runtime#66512, dotnet/runtime#66586, dotnet/runtime#66589, dotnet/runtime#66597, dotnet/runtime#66476, and dotnet/runtime#67125; that significant amount of work means all that code that gets vectorized using these abstractions will light-up on mono and coreclr alike. Second, thanks primarily to dotnet/runtime#70086, mono now knows how to translate `Vector128<T>` operations to WASM's SIMD instruction set, such that code vectorized with `Vector128<T>` will also be accelerated when running in Blazor wasm applications and anywhere else WASM might be executed.

CHAPTER 6

Reflection

Reflection is one of those areas you either love or hate (I find it a bit humorous to be writing this section immediately after writing the Native AOT section). It's immensely powerful, providing the ability to query all of the metadata for code in your process and for arbitrary assemblies you might encounter, to invoke arbitrary functionality dynamically, and even to emit dynamically-generated IL at run-time. It's also difficult to handle well in the face of tooling like a linker or a solution like Native AOT that needs to be able to determine at build time exactly what code will be executed, and it's generally quite expensive at run-time; thus it's both something we strive to avoid when possible but also invest in reducing the costs of, as it's so popular in so many different kinds of applications because it is incredibly useful. As with most releases, it's seen some nice improvements in .NET 7.

One of the most impacted areas is reflection invoke. Available via `MethodBase.Invoke`, this functionality let's you take a `MethodBase` (e.g. `MethodInfo`) object that represents some method for which the caller previously queried, and call it, with arbitrary arguments that the runtime needs to marshal through to the callee, and with an arbitrary return value that needs to be marshaled back. If you know the signature of the method ahead of time, the best way to optimize invocation speed is to create a delegate from the `MethodBase` via `CreateDelegate<T>` and then use that delegate for all future invocations. But in some circumstances, you don't know the signature at compile time, and thus can't easily rely on delegates with known matching signatures. To address this, some libraries have taken to using reflection emit to generate code at run-time specific to the target method. This is extremely complicated and it's not something we want apps to have to do. Instead, in .NET 7 via dotnet/runtime#66357, dotnet/runtime#69575, and dotnet/runtime#74614, `Invoke` will itself use reflection emit (in the form of `DynamicMethod`) to generate a delegate that is customized for invoking the target, and then future invocation via that `MethodInfo` will utilize that generated method. This gives developers most of the performance benefits of a custom reflection emit-based implementation but without having the complexity or challenges of such an implementation in their own code base.

```
private MethodInfo _method;

[GlobalSetup]
public void Setup() => _method = typeof(Program).GetMethod("MyMethod",
BindingFlags.NonPublic | BindingFlags.Static);

[Benchmark]
public void MethodInfoInvoke() => _method.Invoke(null, null);

private static void MyMethod() { }
```

Method	Runtime	Mean	Ratio
MethodInfoInvoke	.NET 6.0	43.846 ns	1.00
MethodInfoInvoke	.NET 7.0	8.078 ns	0.18

Reflection also involves lots of manipulation of objects that represent types, methods, properties, and so on, and tweaks here and there can add up to a measurable difference when using these APIs. For example, I've talked in past performance posts about how, potentially counterintuitively, one of the ways we've achieved performance boosts is by porting native code from the runtime back into managed C#. There are a variety of ways in which doing so can help performance, but one is that there is some overhead associated with calling from managed code into the runtime, and eliminating such hops avoids that overhead. This can be seen in full effect in [dotnet/runtime#71873](), which moves several of these "FCalls" related to `Type`, `RuntimeType` (the `Type`-derived class used by the runtime to represent its types), and `Enum` out of native into managed.

```
[Benchmark]
public Type GetUnderlyingType() => Enum.GetUnderlyingType(typeof(DayOfWeek));
```

Method	Runtime	Mean	Ratio
GetUnderlyingType	.NET 6.0	27.413 ns	1.00
GetUnderlyingType	.NET 7.0	5.115 ns	0.19

Another example of this phenomenon comes in [dotnet/runtime#62866](), which moved much of the underlying support for `AssemblyName` out of native runtime code into managed code in CoreLib. That in turn has an impact on anything that uses it, such as when using `Activator.CreateInstance` overloads that take assembly names that need to be parsed.

```
private readonly string _assemblyName = typeof(MyClass).Assembly.FullName;
private readonly string _typeName = typeof(MyClass).FullName;
public class MyClass { }

[Benchmark]
public object CreateInstance() => Activator.CreateInstance(_assemblyName, _typeName);
```

Method	Runtime	Mean	Ratio
CreateInstance	.NET 6.0	3.827 us	1.00
CreateInstance	.NET 7.0	2.276 us	0.60

Other changes contributed to `Activator.CreateInstance` improvements as well. [dotnet/runtime#67148]() removed several array and list allocations from inside of the `RuntimeType.CreateInstanceImpl` method that's used by `CreateInstance` (using `Type.EmptyTypes` instead of allocating a `new Type[0]`, avoiding unnecessarily turning a builder into an array, etc.), resulting in less allocation and faster throughput.

```
[Benchmark]
public void CreateInstance() => Activator.CreateInstance(typeof(MyClass),
BindingFlags.NonPublic | BindingFlags.Instance, null, Array.Empty<object>(), null);

internal class MyClass
{
```

```
    internal MyClass() { }
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
CreateInstance	.NET 6.0	167.8 ns	1.00	320 B	1.00
CreateInstance	.NET 7.0	143.4 ns	0.85	200 B	0.62

And since we were talking about `AssemblyName`, other PRs improved it in other ways as well. [dotnet/runtime#66750](), for example, updated the computation of `AssemblyName.FullName` to use stack-allocated memory and `ArrayPool<char>` instead of using a `StringBuilder`:

```
private AssemblyName[] _names = AppDomain.CurrentDomain.GetAssemblies().Select(a => new
AssemblyName(a.FullName)).ToArray();

[Benchmark]
public int Names()
{
    int sum = 0;
    foreach (AssemblyName name in _names)
    {
        sum += name.FullName.Length;
    }
    return sum;
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Names	.NET 6.0	3.423 us	1.00	9.14 KB	1.00
Names	.NET 7.0	2.010 us	0.59	2.43 KB	0.27

More reflection-related operations have also been turned into JIT intrinsics, as discussed earlier enabling the JIT to compute answers to various questions at JIT compile time rather than at run-time. This was done, for example, for `Type.IsByRefLike` in [dotnet/runtime#67852]().

```
[Benchmark]
public bool IsByRefLike() => typeof(ReadOnlySpan<char>).IsByRefLike;
```

Method	Runtime	Mean	Ratio	Code Size
IsByRefLike	.NET 6.0	2.1322 ns	1.000	31 B
IsByRefLike	.NET 7.0	0.0000 ns	0.000	6 B

That the .NET 7 version is so close to zero is called out in a warning by benchmarkdotnet:

```
// * Warnings *
ZeroMeasurement
  Program.IsByRefLike: Runtime=.NET 7.0, Toolchain=net7.0 -> The method duration is
indistinguishable from the empty method duration
```

and it's so indistinguishable from an empty method because that's effectively what it is, as we can see from the disassembly:

```
; Program.IsByRefLike()
       mov       eax,1
       ret
; Total bytes of code 6
```

There are also improvements that are hard to see but that remove overheads as part of populating reflection's caches, which end up reducing the work done typically on startup paths, helping apps to launch faster. [dotnet/runtime#66825](), [dotnet/runtime#66912](), and [dotnet/runtime#67149]() all fall into this category by removing unnecessary or duplicative array allocations as part of gathering data on parameters, properties, and events.

Interop

.NET has long had great support for interop, enabling .NET applications to consume huge amounts of functionality written in other languages and/or exposed by the underlying operating system. The bedrock of this support has been "Platform Invoke," or "P/Invoke," represented in code by `[DllImport(...)]` applied to methods. The `DllImportAttribute` enables declaring a method that can be called like any other .NET method but that actually represents some external method that the runtime should call when this managed method is invoked. The DllImport specifies details about in what library the function lives, what its actual name is in the exports from that library, high-level details about marshalling of input arguments and return values, and so on, and the runtime ensures all the right things happen. This mechanism works on all operating systems. For example, Windows has a method `CreatePipe` for creating an anonymous pipe:

```
BOOL CreatePipe(
  [out]           PHANDLE               hReadPipe,
  [out]           PHANDLE               hWritePipe,
  [in, optional]  LPSECURITY_ATTRIBUTES lpPipeAttributes,
  [in]            DWORD                 nSize
);
```

If I want to call this function from C#, I can declare a `[DllImport(...)]` counterpart to it which I can then invoke as I can any other managed method:

```
[DllImport("kernel32", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
private static unsafe extern bool CreatePipe(
    out SafeFileHandle hReadPipe,
    out SafeFileHandle hWritePipe,
    void* lpPipeAttributes,
    uint nSize);
```

There are several interesting things to note here. Several of the arguments are directly blittable with the same representation on the managed and native side of the equation, e.g. `lpPipeAttributes` is a pointer and `nSize` is a 32-bit integer. But what about the return value? The `bool` type in C# (`System.Boolean`) is a one-byte type, but the `BOOL` type in the native signature is four bytes; thus code calling this managed method can't just directly invoke the native function somehow, as there needs to be some "marshalling" logic that converts the four-byte return `BOOL` into the one-byte return `bool`. Simiarly, the native function has two out pointers for `hReadPipe` and `hWritePipe`, but the managed signature declares two `SafeFileHandles` (a `SafeHandle` is a .NET type that wraps a pointer and provides a finalizer and `Dispose` method for ensuring that pointer is appropriately cleaned up when it's no longer being used). Some logic needs to take the output handles generated by the native function and wrap them into these `SafeFileHandles` to be output from the managed method. And what about that `SetLastError = true`? .NET has methods like `Marshal.GetLastPInvokeError()`, and some code somewhere needs to take any error produced by this method and ensure it's available for consumption via a subsequent `GetLastPInvokeError()`.

If there's no marshalling logic required, such that the managed signature and native signature are for all intents and purposes the same, all arguments blittable, all return values blittable, no additional logic required around the invocation of the method, etc., then a `[DllImport(...)]` ends up being a simple passthrough with the runtime needing to do very little work to implement it. If, however, the `[DllImport(...)]` involves any of this marshalling work, the runtime needs to generate a "stub," creating a dedicated method that's called when the `[DllImport(...)]` is called, that handles fixing up all inputs, that delegates to the actual native function, and that fixes up all of the outputs. That stub is generated at execution time, with the runtime effectively doing reflection emit, generating IL dynamically that's then JIT'd.

There are a variety of downsides to this. First, it takes time to generate all that marshalling code, time which can then negatively impact user experience for things like startup. Second, the nature of its implementation inhibits various optimizations, such as inlining. Third, there are platforms that don't allow for JIT'ing due to the security exposure of allowing for dynamically generated code to then be executed (or in the case of Native AOT, where there isn't a JIT at all). And fourth, it's all hidden away making it more challenging for a developer to really understand what's going on.

But what if that logic could all be generated at build time rather than at run time? The cost of generating the code would be incurred only at build time and not on every execution. The code would effectively just end up being user code that has all of the C# compiler's and runtime's optimizations available to it. The code, which then would just be part of the app, would be able to be ahead-of-time compiled using whatever AOT system is desirable, whether it be crossgen or Native AOT or some other system. And the code would be inspectable, viewable by users to understand exactly what work is being done on their behalf. Sounds pretty desirable. Sounds magical. Sounds like a job for a Roslyn source generator, mentioned earlier.

.NET 6 included several source generators in the .NET SDK, and .NET 7 doubles down on this effort including several more. One of these is the brand new LibraryImport generator, which provides exactly the magical, desirable solution we were just discussing.

Let's return to our previous `CreatePipe` example. We'll make two small tweaks. We change the attribute from `DllImport` to `LibraryImport`, and we change the `extern` keyword to be `partial`:

```
[LibraryImport("kernel32", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
private static unsafe partial bool CreatePipe(
    out SafeFileHandle hReadPipe,
    out SafeFileHandle hWritePipe,
    void* lpPipeAttributes,
    uint nSize);
```

Now if you're following along at home in Visual Studio, try right-clicking on CreatePipe and selecting Go to Definition. That might seem a little strange. "Go to Definition? Isn't this the definition?" This is a partial method, which is a way of declaring something that another partial definition fills in, and in this case, a source generator in .NET 7 SDK has noticed this method with the `[LibraryImport]` attribute and fully generated the entire marshalling stub code in C# that's built directly into the assembly. While by default that code isn't persisted, Visual Studio still enables you to browse it (and you can opt-in to having it persisted on disk by adding a `<EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>` property into your .csproj). Here's what it currently looks like for that method:

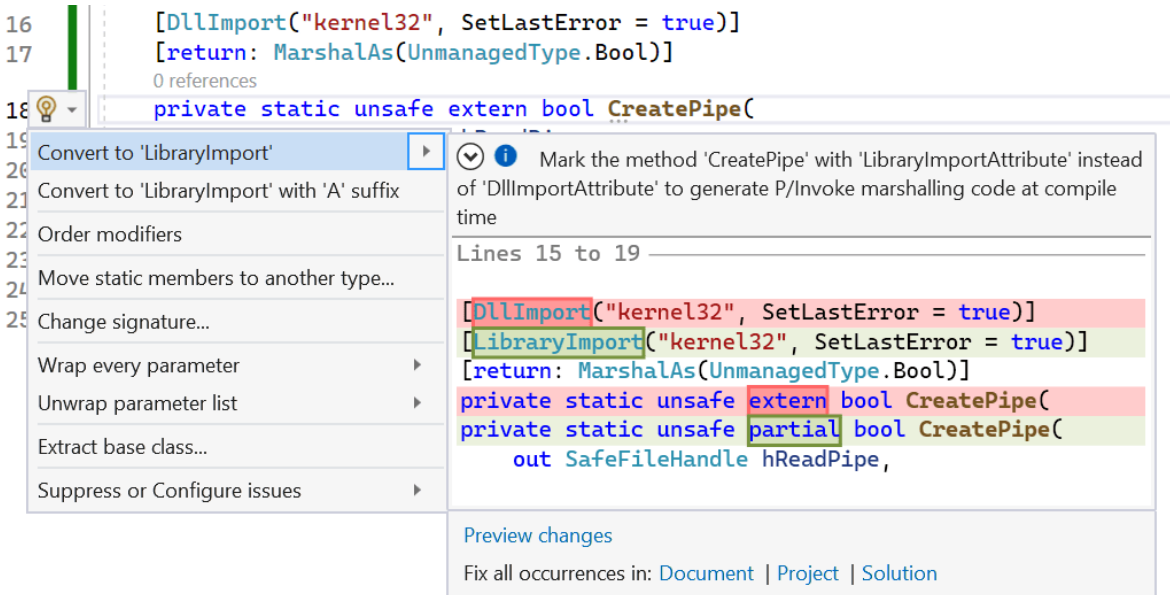
```
[System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Interop.LibraryImportGenerator",
"7.0.6.42316")]
[System.Runtime.CompilerServices.SkipLocalsInitAttribute]
private static unsafe partial bool CreatePipe(out
global::Microsoft.Win32.SafeHandles.SafeFileHandle hReadPipe, out
global::Microsoft.Win32.SafeHandles.SafeFileHandle hWritePipe, void* lpPipeAttributes, uint
nSize)
{
    int __lastError;
    bool __invokeSucceeded = default;
    System.Runtime.CompilerServices.Unsafe.SkipInit(out hReadPipe);
    System.Runtime.CompilerServices.Unsafe.SkipInit(out hWritePipe);
    System.IntPtr __hReadPipe_native = default;
    System.IntPtr __hWritePipe_native = default;
    bool __retVal;
    int __retVal_native = default;

    // Setup - Perform required setup.
    global::Microsoft.Win32.SafeHandles.SafeFileHandle hReadPipe__newHandle = new
global::Microsoft.Win32.SafeHandles.SafeFileHandle();
    global::Microsoft.Win32.SafeHandles.SafeFileHandle hWritePipe__newHandle = new
global::Microsoft.Win32.SafeHandles.SafeFileHandle();
    try
    {
        {
            System.Runtime.InteropServices.Marshal.SetLastSystemError(0);
            __retVal_native = __PInvoke(&__hReadPipe_native, &__hWritePipe_native,
lpPipeAttributes, nSize);
            __lastError = System.Runtime.InteropServices.Marshal.GetLastSystemError();
        }

        __invokeSucceeded = true;

        // Unmarshal - Convert native data to managed data.
        __retVal = __retVal_native != 0;
    }
    finally
    {
        if (__invokeSucceeded)
        {
            // GuaranteedUnmarshal - Convert native data to managed data even in the case
of an exception during the non-cleanup phases.
            System.Runtime.InteropServices.Marshal.InitHandle(hWritePipe__newHandle,
__hWritePipe_native);
            hWritePipe = hWritePipe__newHandle;
            System.Runtime.InteropServices.Marshal.InitHandle(hReadPipe__newHandle,
__hReadPipe_native);
            hReadPipe = hReadPipe__newHandle;
        }
    }

    System.Runtime.InteropServices.Marshal.SetLastPInvokeError(__lastError);
    return __retVal;

    // Local P/Invoke
    [System.Runtime.InteropServices.DllImportAttribute("kernel32", EntryPoint =
```

```csharp
"CreatePipe", ExactSpelling = true)]
    static extern unsafe int __PInvoke(System.IntPtr* hReadPipe, System.IntPtr* hWritePipe,
void* lpPipeAttributes, uint nSize);
}
```

With this, you can read exactly the marshalling work that's being performed. Two `SafeHandle` instances are being allocated and then later after the native function completes, the `Marshal.InitHandle` method is used to store the resulting handles into these instances (the allocations happen before the native function call, as performing them after the native handles have already been produced increases the chances of a leak if the `SafeHandle` allocation fails due to an out-of-memory situation). The `BOOL` to `bool` conversion happens via a `!= 0` comparison. And the error information is captured by calling `Marshal.GetLastSystemError()` just after the native function call and then `Marshal.SetLastPInvokeError(int)` just prior to returning. The actual native function call is still implemented with a `[DllImport(...)]`, but now that P/Invoke is blittable and doesn't require any stub to be generated by the runtime, as all that work has been handled in this C# code.

A sheer ton of work went in to enabling this. I touched on some of it last year in [Performance Improvements in .NET 6](), but a significant amount of additional effort has gone into .NET 7 to polish the design and make the implementation robust, roll it out across all of [dotnet/runtime]() and beyond, and expose the functionality for all C# developers to use:

- The LibraryImport generator started its life as an experiment in [dotnet/runtimelab](). When it was ready, [dotnet/runtime#59579]() brought 180 commits spanning years of effort into the [dotnet/runtime]() main branch.
- In .NET 6, there were almost 3000 `[DllImport]` uses throughout the core .NET libraries. As of my writing this, in .NET 7 there are… let me search… wait for it… 7 (I was hoping I could say 0, but there are just a few stragglers, mostly related to COM interop, still remaining). That's not a transformation that happens over night. A multitude of PRs went library by library converting from the old to the new, such as [dotnet/runtime#62295]() and [dotnet/runtime#61640]() for System.Private.CoreLib, [dotnet/runtime#61742]() and [dotnet/runtime#62309]() for the cryptography libraries, [dotnet/runtime#61765]() for networking, [dotnet/runtime#61996]() and [dotnet/runtime#61638]() for most of the other I/O-related libraries, and a long-tail of additional porting in [dotnet/runtime#61975](), [dotnet/runtime#61389](), [dotnet/runtime#62353](), [dotnet/runtime#61990](), [dotnet/runtime#61949](), [dotnet/runtime#61805](), [dotnet/runtime#61741](), [dotnet/runtime#61184](), [dotnet/runtime#54290](), [dotnet/runtime#62365](), [dotnet/runtime#61609](), [dotnet/runtime#61532](), and [dotnet/runtime#54236]().
- Such porting is significantly easier when there's a tool to help automate it. [dotnet/runtime#72819]() enables the analyzer and fixer for performing these transformations. :::{custom-style=Figure}



:::

There were plenty of other PRs that went into making the LibraryImport generator a reality for .NET 7. To highlight just a few more, dotnet/runtime#63320 introduces a new `[DisabledRuntimeMarshalling]` attribute that can be specified at the assembly level to disable all of the runtime's built-in marshalling; at that point, the only marshalling performed as part of interop is the marshaling done in the user's code, e.g. that which is generated by `[LibraryImport]`. Other PRs like dotnet/runtime#67635 and dotnet/runtime#68173 added new marshaling types that encompass common marshaling logic and can be referenced from `[LibraryImport(...)]` use to customize how marshaling is performed (the generator is pattern-based and allows for customization of marshalling by providing types that implement the right shape, which these types do in support of the most common marshalling needs). Really usefully, dotnet/runtime#71989 added support for marshaling `{ReadOnly}Span<T>`, such that spans can be used directly in `[LibraryImport(...)]` method signatures, just as arrays can be (examples in dotnet/runtime are available in dotnet/runtime#73256). And dotnet/runtime#69043 consolidated logic to be shared between the runtime's marshalling support in `[DllImport]` and the generators support with `[LibraryImport]`.

One more category of interop-related changes that I think are worth talking about are to do with `SafeHandle` cleanup. As a reminder, `SafeHandle` exists to mitigate various issues around managing native handles and file descriptors. A native handle or file descriptor is just a memory address or number that refers to some owned resource and which must be cleaned up / closed when done with it. A `SafeHandle` at its core is just a managed object that wraps such a value and provides a `Dispose` method and a finalizer for closing it. That way, if you neglect to `Dispose` of the `SafeHandle` in order to close the resource, the resource will still be cleaned up when the `SafeHandle` is garbage collected and its finalizer eventually run. `SafeHandle` then also provides some synchronization around that closure, trying to minimize the possibility that the resource is closed while it's still in use. It provides `DangerousAddRef` and `DangerousRelease` methods that increment and decrement a ref count, respectively, and if `Dispose` is called while the ref count is above zero, the actual releasing of the handle triggered by `Dispose` is delayed until the ref count goes back to 0. When you pass a `SafeHandle` into a P/Invoke, the generated code for that P/Invoke handles calling `DangerousAddRef` and `DangerousRelease` (and due to the wonders of LibraryImport I've already extolled, you can easily see that being done, such as in the previous generated code example). Our code tries hard to clean up after `SafeHandle`s deterministically, but it's quite easy to accidentally leave some for finalization.

[dotnet/runtime#71854](https://github.com/dotnet/runtime/pull/71854) added some debug-only tracking code to `SafeHandle` to make it easier for developers working in [dotnet/runtime](https://github.com/dotnet/runtime) (or more specifically, developers using a checked build of the runtime) to find such issues. When the `SafeHandle` is constructed, it captures the current stack trace, and if the `SafeHandle` is finalized, it dumps that stack trace to the console, making it easy to see where `SafeHandles` that do end up getting finalized were created, in order to track them down and ensure they're being disposed of. As is probably evident from that PR touching over 150 files and almost 1000 lines of code, there were quite a few places that benefited from clean up. Now to be fair, many of these are on exceptional code paths. For example, consider a hypothetical P/Invoke like:

```
[LibraryImport("SomeLibrary", SetLastError = true)]
internal static partial SafeFileHandle CreateFile();
```

and code that uses it like:

```
SafeFileHandle handle = Interop.CreateFile();
if (handle.IsInvalid)
{
    throw new UhOhException(Marshal.GetLastPInvokeError());
}
return handle;
```

Seems straightforward enough. Except this code will actually leave a `SafeHandle` for finalization on the failure path. It doesn't matter that `SafeHandle` has an invalid handle in it, it's still a finalizable object. To deal with that, this code would have been more robustly written as:

```
SafeFileHandle handle = Interop.CreateFile();
if (handle.IsInvalid)
{
    int lastError = Marshal.GetLastPInvokeError();
    handle.Dispose(); // or handle.SetHandleAsInvalid()
    throw new UhOhException(lastError);
}
return handle;
```

That way, this `SafeHandle` won't create finalization pressure even in the case of failure. Note, as well, that as part of adding in the `Dispose` call, I also moved the `Marshal.GetLastPInvokeError()` up. That's because calling `Dispose` on a `SafeHandle` may end up invoking the `SafeHandle`'s `ReleaseHandle` method, which the developer of the `SafeHandle`-derived type will have overridden to close the resource, which typically involves making another P/Invoke. And if that P/Invoke has `SetLastError=true` on it, it can overwrite the very error code for which we're about to throw. Hence, we access and store the last error immediately after the interop call once we know it failed, then clean up, and only then throw. All that said, there were a myriad of places in that PR where `SafeHandles` were being left for finalization even on the success path. And that PR wasn't alone. [dotnet/runtime#71991](https://github.com/dotnet/runtime/pull/71991), [dotnet/runtime#71854](https://github.com/dotnet/runtime/pull/71854), [dotnet/runtime#72116](https://github.com/dotnet/runtime/pull/72116), [dotnet/runtime#72189](https://github.com/dotnet/runtime/pull/72189), [dotnet/runtime#72222](https://github.com/dotnet/runtime/pull/72222), [dotnet/runtime#72203](https://github.com/dotnet/runtime/pull/72203), and [dotnet/runtime#72279](https://github.com/dotnet/runtime/pull/72279) all found and fixed many occurrences of `SafeHandles` being left for finalization (many thanks to the diagnostics put in place in the earlier mentioned PR).

Other PRs also accrued to improved interop performance. dotnet/runtime#70000 from [@huoyaoyuan](https://github.com/huoyaoyuan) rewrote several delegate-related "FCalls" from being implemented in native code to instead being managed, resulting in less overhead when invoking these operations that are commonly involved in scenarios involving `Marshal.GetDelegateForFunctionPointer`. dotnet/runtime#68694 also moved some trivial functionality from native to managed, as part of relaxing argument validation on the use of pinning handles. This in turn measurably reduced the overhead involved with using `GCHandle.Alloc` for such pinning handles:

```
private byte[] _buffer = new byte[1024];

[Benchmark]
public void PinUnpin()
{
    GCHandle.Alloc(_buffer, GCHandleType.Pinned).Free();
}
```

Method	Runtime	Mean	Ratio	Code Size
PinUnpin	.NET 6.0	37.11 ns	1.00	353 B
PinUnpin	.NET 7.0	32.17 ns	0.87	232 B

Threading

Threading is one of those cross-cutting concerns that impacts every application, such that changes in the threading space can have a wide-spread impact. This release sees two very substantial changes to the `ThreadPool` itself; [dotnet/runtime#64834](https://github.com/dotnet/runtime/pull/64834) switches the "IO pool" over to using an entirely managed implementation (whereas previously the IO pool was still in native code even though the worker pool had been moved entirely to managed in previous releases), and [dotnet/runtime#71864](https://github.com/dotnet/runtime/pull/71864) similarly switches the timer implementation from one based in native to one entirely in managed code. Those two changes can impact performance, and the former was demonstrated to on larger hardware, but for the most part that wasn't their primary goal. Instead, other PRs have been focused on improving throughput.

One in particular is [dotnet/runtime#69386](https://github.com/dotnet/runtime/pull/69386). The `ThreadPool` has a "global queue" that any thread can queue work into, and then each thread in the pool has its own "local queue" (which any thread can dequeue from but only the owning thread can enqueue into). When a worker needs another piece of work to process, it first checks its own local queue, then it checks the global queue, and then only if it couldn't find work in either of those two places, it goes and checks all of the other threads' local queues to see if it can help lighten their load. As machines scale up to have more and more cores, and more and more threads, there's more and more contention on these shared queues, and in particular on the global queue. This PR addresses this for such larger machines by introducing additional global queues once the machine reaches a certain threshold (32 processors today). This helps to partition accesses across multiple queues, thereby decreasing contention.

Another is [dotnet/runtime#57885](https://github.com/dotnet/runtime/pull/57885). In order to coordinate threads, when work items were enqueued and dequeued, the pool was issuing requests to its threads to let them know that there was work available to do. This, however, often resulted in oversubscription, where more threads than necessary would race to try to get work items, especially when the system wasn't at full load. That in turn would manifest as a throughput regression. This change overhauls how threads are requested, such that only one additional thread is requested at a time, and after that thread has dequeued its first work item, it can issue a request for an additional thread if there's work remaining, and then that one can issue an additional request, and so on. Here's one of our performance tests in our performance test suite (I've simplified it down to remove a bunch of configuration options from the test, but it's still accurately one of those configurations). At first glance you might think, "hey, this is a performance test about `ArrayPool`, why is it showing up in a threading discussion?" And, you'd be right, this is a performance test that was written focused on `ArrayPool`. However, as mentioned earlier, threading impacts everything, and in this case, that `await Task.Yield()` in the middle there causes the remainder of this method to be queued to the `ThreadPool` for execution. And because of how the test is structured, doing "real work" that competes for CPU cycles with thread pool threads all racing to get their next task, it shows a measurable improvement when moving to .NET 7.

```
private readonly byte[][] _nestedArrays = new byte[8][];
private const int Iterations = 100_000;

private static byte IterateAll(byte[] arr)
{
    byte ret = default;
    foreach (byte item in arr) ret = item;
    return ret;
}

[Benchmark(OperationsPerInvoke = Iterations)]
public async Task MultipleSerial()
{
    for (int i = 0; i < Iterations; i++)
    {
        for (int j = 0; j < _nestedArrays.Length; j++)
        {
            _nestedArrays[j] = ArrayPool<byte>.Shared.Rent(4096);
            _nestedArrays[j].AsSpan().Clear();
        }

        await Task.Yield();

        for (int j = _nestedArrays.Length - 1; j >= 0; j--)
        {
            IterateAll(_nestedArrays[j]);
            ArrayPool<byte>.Shared.Return(_nestedArrays[j]);
        }
    }
}
```

Method	Runtime	Mean	Ratio
MultipleSerial	.NET 6.0	14.340 us	1.00
MultipleSerial	.NET 7.0	9.262 us	0.65

There have been improvements outside of `ThreadPool`, as well. One notable change is in the handling of `AsyncLocal<T>`s, in [dotnet/runtime#68790](https://github.com/dotnet/runtime/pull/68790). `AsyncLocal<T>` is integrated tightly with `ExecutionContext`; in fact, in .NET Core, `ExecutionContext` is *entirely* about flowing `AsyncLocal<T>` instances. An `ExecutionContext` instance maintains a single field, a map data structure, that stores the data for all `AsyncLocal<T>` with data present in that context. Each `AsyncLocal<T>` has an object it uses as a key, and any gets or sets on that `AsyncLocal<T>` manifest as getting the current `ExecutionContext`, looking up that `AsyncLocal<T>`'s key in the context's dictionary, and then either returning whatever data it finds, or in the case of a setter, creating a new `ExecutionContext` with an updated dictionary and publishing that back. This dictionary thus needs to be very efficient for reads and writes, as developers expect `AsyncLocal<T>` access to be as fast as possible, often treating it as if it were any other local. So, to optimize these lookups, the representation of that dictionary changes based on how many `AsyncLocal<T>`s are represented in this context. For up to three items, dedicated implementations with fields for each of the three keys and values were used. Above that up to around 16 elements, an array of key/value pairs was used. And above that, a `Dictionary<,>` was used. For the most part, this has worked well, with the majority of `ExecutionContext`s being able to represent many flows with one of the first three types. However, it turns out that four active `AsyncLocal<T>` instances is really common, especially in ASP.NET where ASP.NET infrastructure itself uses a couple.

So, this PR took the complexity hit to add a dedicated type for four key/value pairs, in order to optimize from one to four of them rather than one to three. While this improves throughput a bit, its main intent was to improve allocation, which is does over .NET 6 by ~20%.

```
private AsyncLocal<int> asyncLocal1 = new AsyncLocal<int>();
private AsyncLocal<int> asyncLocal2 = new AsyncLocal<int>();
private AsyncLocal<int> asyncLocal3 = new AsyncLocal<int>();
private AsyncLocal<int> asyncLocal4 = new AsyncLocal<int>();

[Benchmark(OperationsPerInvoke = 4000)]
public void Update()
{
    for (int i = 0; i < 1000; i++)
    {
        asyncLocal1.Value++;
        asyncLocal2.Value++;
        asyncLocal3.Value++;
        asyncLocal4.Value++;
    }
}
```

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
Update	.NET 6.0	61.96 ns	1.00	1,272 B	176 B	1.00
Update	.NET 7.0	61.92 ns	1.00	1,832 B	144 B	0.82

Another valuable fix comes for locking in [dotnet/runtime#70165](). This particular improvement is a bit harder to demonstrate with benchmarkdotnet, so just try running this program, first on .NET 6 and then on .NET 7:

```
using System.Diagnostics;

var rwl = new ReaderWriterLockSlim();
var tasks = new Task[100];
int count = 0;

DateTime end = DateTime.UtcNow + TimeSpan.FromSeconds(10);
while (DateTime.UtcNow < end)
{
    for (int i = 0; i < 100; ++i)
    {
        tasks[i] = Task.Run(() =>
        {
            var sw = Stopwatch.StartNew();
            rwl.EnterReadLock();
            rwl.ExitReadLock();
            sw.Stop();
            if (sw.ElapsedMilliseconds >= 10)
            {
                Console.WriteLine(Interlocked.Increment(ref count));
            }
        });
    }

    Task.WaitAll(tasks);
}
```

This is simply spinning up 100 tasks, each of which enters and exits a read-write lock, waits for them all, and then does the process over again, for 10 seconds. It also times how long it takes to enter and exit the lock, and writes a warning if it had to wait for at least 15ms. When I run this on .NET 6, I get ~100 occurrences of it taking >= 10 ms to enter/exit the lock. On .NET 7, I get 0 occurrences. Why the difference? The implementation of `ReaderWriterLockSlim` has its own spin loop implementation, and that spin loop tries to mix together various things to do as it spins, ranging from calling `Thread.SpinWait` to `Thread.Sleep(0)` to `Thread.Sleep(1)`. The issue lies in the `Thread.Sleep(1)`. That's saying "put this thread to sleep for 1 millisecond"; however, the operating system has the ultimate say on such timings, and on Windows, by default that sleep is going to be closer to 15 milliseconds (on Linux it's a bit lower but still quite high). Thus, every time there was enough contention on the lock to force it to call `Thread.Sleep(1)`, we'd incur a delay of at least 15 milliseconds, if not more. The aforementioned PR fixed this by eliminating use of `Thread.Sleep(1)`.

One final threading-related change to call out: [dotnet/runtime#68639](). This one is Windows specific. Windows has the concept of processor groups, each of which can have up to 64 cores in it, and by default when a process runs, it's assigned a specific processor group and can only use the cores in that group. With .NET 7, the runtime flips its default so that by default it will try to use all processor groups if possible.

Primitive Types and Numerics

We've looked at code generation and GC, at threading and vectorization, at interop… let's turn our attention to some of the fundamental types in the system. Primitives like `int` and `bool` and `double`, core types like `Guid` and `DateTime`, they form the backbone on which everything is built, and every release it's exciting to see the improvements that find their way into these types.

`float` and `double` got a very nice boost in their implementation of parsing (e.g. `double.Parse`, `float.TryParse`, etc.). [dotnet/runtime#62301]() from [@CarlVerret](https://github.com/CarlVerret) significantly improves `double.Parse` and `float.Parse` for parsing UTF16 text into floating-point values. This is particularly neat because it's based on some [relatively recent research]() from [@lemire](https://github.com/lemire) and [@CarlVerret](https://github.com/CarlVerret), who used C# with .NET 5 to implement a very fast implementation for parsing floating-point numbers, and that implementation how now found its way into .NET 7!

```
private string[] _valuesToParse;

[GlobalSetup]
public void Setup()
{
    using HttpClient hc = new HttpClient();
    string text =
hc.GetStringAsync("https://raw.githubusercontent.com/CarlVerret/csFastFloat/1d800237275f759
b743b86fcce6680d072c1e834/Benchmark/data/canada.txt").Result;
    var lines = new List<string>();
    foreach (ReadOnlySpan<char> line in text.AsSpan().EnumerateLines())
    {
        ReadOnlySpan<char> trimmed = line.Trim();
        if (!trimmed.IsEmpty)
        {
            lines.Add(trimmed.ToString());
        }
    }
    _valuesToParse = lines.ToArray();
}

[Benchmark]
public double ParseAll()
{
    double total = 0;
    foreach (string s in _valuesToParse)
    {
        total += double.Parse(s);
```

```
    }
    return total;
}
```

Method	Runtime	Mean	Ratio
ParseAll	.NET 6.0	26.84 ms	1.00
ParseAll	.NET 7.0	12.63 ms	0.47

`bool.TryParse` and `bool.TryFormat` were also improved. [dotnet/runtime#64782](https://github.com/dotnet/runtime/pull/64782) streamlined these implementations by using `BinaryPrimitives` to perform fewer writes and reads. For example, instead of `TryFormat` writing out "True" by doing:

```
destination[0] = 'T';
destination[1] = 'r';
destination[2] = 'u';
destination[3] = 'e';
```

which requires four writes, it can instead implement the same operation in a single write by doing:

```
BinaryPrimitives.WriteUInt64LittleEndian(MemoryMarshal.AsBytes(destination),
0x65007500720054); // "True"
```

That `0x65007500720054` is the numerical value of the four characters in memory as a single `ulong`. You can see the impact of these changes with a microbenchmark:

```
private bool _value = true;
private char[] _chars = new char[] { 'T', 'r', 'u', 'e' };

[Benchmark] public bool ParseTrue() => bool.TryParse(_chars, out _);
[Benchmark] public bool FormatTrue() => _value.TryFormat(_chars, out _);
```

Method	Runtime	Mean	Ratio
ParseTrue	.NET 6.0	7.347 ns	1.00
ParseTrue	.NET 7.0	2.327 ns	0.32
FormatTrue	.NET 6.0	3.030 ns	1.00
FormatTrue	.NET 7.0	1.997 ns	0.66

`Enum` gets several performance boosts, as well. For example, when performing an operation like `Enum.IsDefined`, `Enum.GetName`, or `Enum.ToString`, the implementation consults a cache of all of the values defined on the enum. This cache includes the string name and the value for every defined enumeration in the `Enum`. It's also sorted by value in an array, so when one of these operations is performed, the code uses `Array.BinarySearch` to find the index of the relevant entry. The issue with that is one of overheads. When it comes to algorithmic complexity, a binary search is faster than a linear search; after all, a binary search is `O(log N)` whereas a linear search is `O(N)`. However, there's also less overhead for every step of the algorithm in a linear search, and so for smaller values of `N`, it can be much faster to simply do the simple thing. That's what [dotnet/runtime#57973](https://github.com/dotnet/runtime/pull/57973) does for enums. For enums with less than or equal to 32 defined values, the implementation now just does a linear search via the internal `SpanHelpers.IndexOf` (the worker routine behind `IndexOf` on spans, strings, and arrays), and for enums with more than that, it does a `SpanHelpers.BinarySearch` (which is the implementation for `Array.BinarySearch`).

```
private DayOfWeek[] _days = Enum.GetValues<DayOfWeek>();

[Benchmark]
public bool AllDefined()
{
    foreach (DayOfWeek day in _days)
    {
        if (!Enum.IsDefined(day))
        {
            return false;
        }
    }

    return true;
}
```

Method	Runtime	Mean	Ratio
AllDefined	.NET 6.0	159.28 ns	1.00
AllDefined	.NET 7.0	94.86 ns	0.60

`Enum`s also get a boost in conjunction with `Nullable<T>` and `EqualityComparer<T>.Default`. `EqualityComparer<T>.Default` caches a singleton instance of an `EqualityComparer<T>` instance returned from all accesses to `Default`. That singleton is initialized based on the `T` in question, with the implementation choosing from a multitude of different internal implementations, for example a `ByteArrayComparer` specialized for bytes, a `GenericEqualityComparer<T>` for `T`s that implement `IComparable<T>`, and so on. The catch-all, for arbitrary types, is an `ObjectEqualityComparer<T>`. As it happens, nullable enums would end up hitting this catch-all path, which means that every `Equals` call would box the arguments. dotnet/runtime#68077 fixes this by ensuring nullable enums get mapped to (an existing) specialized comparer for `Nullable<T>` and simple tweaks its definition to ensure it can play nicely with enums. The results highlight just how much unnecessary overhead there was previously.

```
private DayOfWeek?[] _enums = Enum.GetValues<DayOfWeek>().Select(e =>
(DayOfWeek?)e).ToArray();

[Benchmark]
[Arguments(DayOfWeek.Saturday)]
public int FindEnum(DayOfWeek value) => IndexOf(_enums, value);

private static int IndexOf<T>(T[] values, T value)
{
    for (int i = 0; i < values.Length; i++)
    {
        if (EqualityComparer<T>.Default.Equals(values[i], value))
        {
            return i;
        }
    }

    return -1;
}
```

Method	Runtime	Mean	Ratio
FindEnum	.NET 6.0	421.608 ns	1.00
FindEnum	.NET 7.0	5.466 ns	0.01

Not to be left out, `Guid`'s equality operations also get faster, thanks to [dotnet/runtime#66889](https://github.com/dotnet/runtime/pull/66889) from [@madelson](https://github.com/madelson). The previous implementation of `Guid` split the data into four 32-bit values and performed 4 `int` comparisons. With this change, if the current hardware has 128-bit SIMD support, the implementation loads the data from the two guids as two vectors and simply does a single comparison.

```
private Guid _guid1 = Guid.Parse("0aa2511d-251a-4764-b374-4b5e259b6d9a");
private Guid _guid2 = Guid.Parse("0aa2511d-251a-4764-b374-4b5e259b6d9a");

[Benchmark]
public bool GuidEquals() => _guid1 == _guid2;
```

Method	Runtime	Mean	Ratio	Code Size
GuidEquals	.NET 6.0	2.119 ns	1.00	90 B
GuidEquals	.NET 7.0	1.354 ns	0.64	78 B

`DateTime` equality is also improved. [dotnet/runtime#59857](https://github.com/dotnet/runtime/pull/59857) shaves some overhead off of `DateTime.Equals`. `DateTime` is implemented with a single `ulong _dateData` field, where the majority of the bits store a ticks offset from 1/1/0001 12:00am and where each tick is 100 nanoseconds, and where the top two bits describe the `DateTimeKind`. Thus the public `Ticks` property returns the value of `_dateData` but with the top two bits masked out, e.g. `_dateData & 0x3FFFFFFFFFFFFFFF`. The equality operators were all then just comparing one `DateTime`'s `Ticks` against the others, such that we effectively get `(dt1._dateData & 0x3FFFFFFFFFFFFFFF) == (dt2._dateData & 0x3FFFFFFFFFFFFFFF)`. However, as a micro-optimization that can instead be expressed more efficiently as `((dt1._dateData ^ dt2._dateData) << 2) == 0`. It's difficult to measure the difference in such tiny operations, but you can see it simply from the number of instructions involved, where on .NET 6 this produces:

```
; Program.DateTimeEquals()
       mov       rax,[rcx+8]
       mov       rdx,[rcx+10]
       mov       rcx,0FFFFFFFFFFFF
       and       rax,rcx
       and       rdx,rcx
       cmp       rax,rdx
       sete      al
       movzx     eax,al
       ret
; Total bytes of code 34
```

and on .NET 7 this produces:

```
; Program.DateTimeEquals()
       mov       rax,[rcx+8]
       mov       rdx,[rcx+10]
       xor       rax,rdx
       shl       rax,2
```

```
       sete      al
       movzx     eax,al
       ret
; Total bytes of code 22
```

so instead of a `mov`, `and`, `and`, and `cmp`, we get just an `xor` and a `shl`.

Other operations on `DateTime` also become more efficient, thanks to [dotnet/runtime#72712](https://github.com/dotnet/runtime/pull/72712) from [@SergeiPavlov](https://github.com/SergeiPavlov) and [dotnet/runtime#73277](https://github.com/dotnet/runtime/pull/73277) from [@SergeiPavlov](https://github.com/SergeiPavlov). In another case of .NET benefiting from recent advancements in research, these PRs implemented the algorithm from Neri and Schneider's "Euclidean Affine Functions and Applications to Calendar Algorithms" in order to improve `DateTime.Day`, `DateTime.DayOfYear`, `DateTime.Month`, and `DateTime.Year`, as well as the internal helper `DateTime.GetDate()` that's used by a bunch of other methods like `DateTime.AddMonths`, `Utf8Formatter.TryFormat(DateTime, ...)`, `DateTime.TryFormat`, and `DateTime.ToString`.

```
private DateTime _dt = DateTime.UtcNow;
private char[] _dest = new char[100];

[Benchmark] public int Day() => _dt.Day;
[Benchmark] public int Month() => _dt.Month;
[Benchmark] public int Year() => _dt.Year;
[Benchmark] public bool TryFormat() => _dt.TryFormat(_dest, out _, "r");
```

Method	Runtime	Mean	Ratio
Day	.NET 6.0	5.2080 ns	1.00
Day	.NET 7.0	2.0549 ns	0.39
Month	.NET 6.0	4.1186 ns	1.00
Month	.NET 7.0	2.0945 ns	0.51
Year	.NET 6.0	3.1422 ns	1.00
Year	.NET 7.0	0.8200 ns	0.26
TryFormat	.NET 6.0	27.6259 ns	1.00
TryFormat	.NET 7.0	25.9848 ns	0.94

So, we've touched on improvements to a few types, but the pièce de résistance around primitive types in this release is "generic math," which impacts almost every primitive type in .NET. There are significant improvements here, some which have been in the making for literally over a decade.

There's an excellent blog post from June dedicated just to generic math, so I won't go into much depth here. At a high level, however, there are now over 30 new interfaces that utilize the new C# 11 static abstract interface methods functionality, exposing wide-ranging operations from exponentiation functions to trigonometric functions to standard numerical operators, all available via generics, such that you can write one implementation that operates over these interfaces generically and have your code applied to any types that implement the interfaces… which all of the numerical types in .NET 7 do (including not just the primitives but also, for example, `BigInteger` and `Complex`). A preview version of this feature, including necessary runtime support, language syntax, C# compiler support, generic interfaces, and interface implementations all shipped in .NET 6 and C# 10, but it wasn't supported for production use, and you had to download an experimental reference assembly in order to get access. With dotnet/runtime#65731, all of this support moved into .NET 7 as supported functionality. dotnet/runtime#66748, dotnet/runtime#67453, dotnet/runtime#69391, dotnet/runtime#69582, dotnet/runtime#69756, and dotnet/runtime#71800 all updated the design and implementation based on feedback from usage in .NET 6 and .NET 7 previews as well as a proper API review with our API review team (a process every new API in .NET goes through before it's shipped publicly). dotnet/runtime#67714 added support for user-defined `checked` operators, a new C# 11 feature that enables both `unchecked` and `checked` variations of operators to be exposed, with the compiler picking the right one based on the `checked` context. dotnet/runtime#68096 also added support for the new C# 11 unsigned right shift operator (`>>>`). And dotnet/runtime#69651, dotnet/runtime#67939, dotnet/runtime#73274, dotnet/runtime#71033, dotnet/runtime#71010, dotnet/runtime#68251, dotnet/runtime#68217, and dotnet/runtime#68094 all added large swaths of new public surface area for various operations, all with highly-efficient managed implementations, in many cases based on the open source AMD Math Library.

While this support is all primarily intended for external consumers, the core libraries do consume some of it internally. You can see how these APIs clean up consuming code even while maintaining performance in PRs like dotnet/runtime#68226 and dotnet/runtime#68183, which use the interfaces to deduplicate a bunch of LINQ code in `Enumerable.Sum`/`Average`/`Min`/`Max`. There are multiple overloads of these methods for `int`, `long`, `float`, `double`, and `decimal`. The GitHub summary of the diffs tells the story on how much code was able to be deleted:

Another simple example comes from the new `System.Formats.Tar` library in .NET 7, which as the name suggests is used for reading and writing archives in any of multiple tar file formats. The tar file formats include integer values in octal representation, so the `TarReader` class needs to parse octal values. Some of these values are 32-bit integers, and some are 64-bit integers. Rather than have two separate `ParseOctalAsUInt32` and `ParseOctalAsUInt64` methods, dotnet/runtime#74281] consolidated the methods into a single `ParseOctal<T>` with the constraint `where T : struct, INumber<T>`. The implementation is then entirely in terms of `T` and can be used for either of these types (plus any other types meeting the constraints, should that ever be needed). What's particularly interesting about this example is the `ParseOctal<T>` method includes use of `checked`, e.g. `value = checked((value * octalFactor) + T.CreateTruncating(digit));`. This is only possible because C# 11 includes the aforementioned support for user-defined checked operators, enabling the generic math interfaces to support both the normal and checked varieties, e.g. the `IMultiplyOperators<,,>` interface contains these methods:

```
static abstract TResult operator *(TSelf left, TOther right);
static virtual TResult operator checked *(TSelf left, TOther right) => left * right;
```

and the compiler will pick the appropriate one based on the context.

In addition to all the existing types that get these interfaces, there are also new types. dotnet/runtime#69204 adds the new `Int128` and `UInt128` types. As these types implement all of the relevant generic math interfaces, they come complete with a huge number of methods, over 100 each, all of which are implemented efficiently in managed code. In the future, the aim is that some set of these will be optimized further by the JIT and to take advantage of hardware acceleration.

Several PRs moved native implementations of these kinds of math operations to managed code. dotnet/runtime#63881 from [@am11](https://github.com/am11) did so for `Math.Abs` and `Math.AbsF` (absolute value), and dotnet/runtime#56236 from [@alexcovington](https://github.com/alexcovington) did so for `Math.ILogB` and `MathF.ILogB` (base 2 integer logarithm). The latter's implementation is based on the MUSL libc implementation of the same algorithm, and in addition to improving performance (in part by avoiding the transition between managed and native code, in part by the actual algorithm employed), it also enabled deleting two distinct implementations from native code, one from the coreclr side and one from the mono side, which is always a nice win from a maintainability perspective.

```
[Benchmark]
[Arguments(12345.6789)]
public int ILogB(double arg) => Math.ILogB(arg);
```

Method	Runtime	arg	Mean	Ratio
ILogB	.NET 6.0	12345.6789	4.056 ns	1.00
ILogB	.NET 7.0	12345.6789	1.059 ns	0.26

Other math operations were also improved in various ways. `Math{F}.Truncate` was improved in dotnet/runtime#65014 from [@MichalPetryka](https://github.com/MichalPetryka) by making it into a JIT intrinsic, such that on Arm64 the JIT could directly emit a `frintz` instruction. dotnet/runtime#65584 did the same for `Max` and `Min` so that the Arm-specific `fmax` and `fmin` instructions could be used. And several `BitConverter` APIs were also turned into intrinsics in dotnet/runtime#71567 in order to enable better code generation in some generic math scenarios.

dotnet/runtime#55121 from [@key-moon](https://github.com/key-moon) also improves parsing, but for `BigInteger`, and more specifically for really, really big `BigIntegers`. The algorithm previously employed for parsing a string into a `BigInteger` was `O(N^2)` where `N` is the number of digits, but while a larger algorithmic complexity than we'd normally like, it has a low constant overhead and so is still reasonable for reasonably-sized values. In contrast, an alternative algorithm is available that runs in `O(N * (log N)^2)` time, but with a much higher constant factor involved. That makes is so that it's really only worth switching for really big numbers. Which is what this PR does. It implements the alternative algorithm and switches over to it when the input is at least 20,000 digits (so, yes, big). But for such large numbers, it makes a significant difference.

```
private string _input = string.Concat(Enumerable.Repeat("1234567890", 100_000)); // "One
miiilliiiion digits"

[Benchmark]
public BigInteger Parse() => BigInteger.Parse(_input);
```

Method	Runtime	Mean	Ratio
Parse	.NET 6.0	3.474 s	1.00
Parse	.NET 7.0	1.672 s	0.48

Also related to `BigInteger` (and not just for really big ones), [dotnet/runtime#35565](https://github.com/dotnet/runtime/pull/35565) from [@sakno](https://github.com/sakno) overhauled much of the internals of `BigInteger` to be based on spans rather than arrays. That in turn enabled a fair amount of use of stack allocation and slicing to avoid allocation overheads, while also improving reliability and safety by moving some code away from unsafe pointers to safe spans. The primary performance impact is visible in allocation numbers, and in particular for operations related to division.

```
private BigInteger _bi1 = BigInteger.Parse(string.Concat(Enumerable.Repeat("9876543210",
100)));
private BigInteger _bi2 = BigInteger.Parse(string.Concat(Enumerable.Repeat("1234567890",
100)));
private BigInteger _bi3 = BigInteger.Parse(string.Concat(Enumerable.Repeat("12345", 10)));

[Benchmark]
public BigInteger ModPow() => BigInteger.ModPow(_bi1, _bi2, _bi3);
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
ModPow	.NET 6.0	1.527 ms	1.00	706 B	1.00
ModPow	.NET 7.0	1.589 ms	1.04	50 B	0.07

Arrays, Strings, and Spans

While there are many forms of computation that can consume resources in applications, some of the most common include processing of data stored in arrays, strings, and now spans. Thus you see a focus in every .NET release on removing as much overhead as possible from such scenarios, while also finding ways to further optimize the concrete operations developers are commonly performing.

Let's start with some new APIs that can help make writing more efficient code easier. When examining string parsing/processing code, it's very common to see characters examined for their inclusion in various sets. For example, you might see a loop looking for characters that are ASCII digits:

```
while (i < str.Length)
{
    if (str[i] >= '0' && str[i] <= '9')
    {
        i++;
    }
}
```

or that are ASCII letters:

```
while (i < str.Length)
{
    if ((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A' && str[i] <= 'Z'))
    {
        i++;
    }
}
```

or other such groups. Interestingly, there's wide-spread variation in how such checks are coded, often depending on how much effort a developer put in to optimizing them, or in some cases likely not even recognizing that some amount of performance was being left on the table. For example, that same ASCII letter check could instead be written as:

```
while (i < str.Length)
{
    if ((uint)((c | 0x20) - 'a') <= 'z' - 'a')
    {
        i++;
    }
}
```

which while more "intense" is also much more concise and more efficient. It's taking advantage of a few tricks. First, rather than having two comparisons to determine whether the character is greater than or equal to the lower bound and less than or equal to the upper bound, it's doing a single comparison based on the distance between the character and the lower bound (`(uint)(c - 'a')`). If `'c'` is beyond `'z'`, then `'c' - 'a'` will be larger than 25, and the comparison will fail. If `'c'` is earlier than `'a'`, then `'c' - 'a'` will be negative, and casting it to `uint` will then cause it to wrap around to a massive number, also larger than 25, again causing the comparison to fail. Thus, we're able to pay a single additional subtraction to avoid an entire additional comparison and branch, which is *almost* always a good deal. The second trick is that `| 0x20`. The ASCII table has some well-thought-out relationships, including that upper-case `'A'` and lower-case `'a'` differ by only a single bit (`'A'` is `0b1000001` and `'a'` is `0b1100001`). To go from any lowercase ASCII letter to its uppercase ASCII equivalent, we thus need only to `& ~0x20` (to turn off that bit), and to go in the opposite direction from any uppercase ASCII letter to its lowercase ASCII equivalent, we need only to `| 0x20` (to turn on that bit). We can take advantage of this in our range check, then, by normalizing our char `c` to be lowercase, such that for the low cost of a bit twiddle, we can achieve both the lowercase and uppercase range checks. Of course, those tricks aren't something we want every developer to have to know and write on each use. Instead, .NET 7 exposes a bunch of new helpers on `System.Char` to encapsulate these common checks, done in an efficient manner. `char` already had methods like `IsDigit` and `IsLetter`, which provided the more comprehensive Unicode meaning of those monikers (e.g. there are ~320 Unicode characters categorized as "digits"). Now in .NET 7, there are also these helpers:

- IsAsciiDigit
- IsAsciiHexDigit
- IsAsciiHexDigitLower
- IsAsciiHexDigitUpper
- IsAsciiLetter
- IsAsciiLetterLower
- IsAsciiLetterUpper
- IsAsciiLetterOrDigit

These methods were added by [dotnet/runtime#69318](), which also employed them in dozens of locations where such checks were being performed across [dotnet/runtime]() (many of them using less-efficient approaches).

Another new API focused on encapsulating a common pattern is the new `MemoryExtensions.CommonPrefixLength` method, introduced by [dotnet/runtime#67929](). This accepts either two `ReadOnlySpan<T>` instances or a `Span<T>` and a `ReadOnlySpan<T>`, and an optional `IEqualityComparer<T>`, and returns the number of elements that are the same at the beginning of each input span. This is useful when you want to know the first place that two inputs differ. [dotnet/runtime#68210]() from [@gfoidl](https://github.com/gfoidl) then utilized the new `Vector128` functionality to provide a basic vectorization of the implementation. As it's comparing two sequences and looking for the first place they differ, this implementation uses a neat trick, which is to have a single method implemented to compare the sequences as bytes. If the `T` being compared is bitwise-equatable and no custom equality comparer is supplied, then it reinterpret-casts the refs from the spans as `byte` refs, and uses the single shared implementation.

Yet another new set of APIs are the `IndexOfAnyExcept` and `LastIndexOfAnyExcept` methods, introduced by [dotnet/runtime#67941]() and used in a variety of additional call sites by [dotnet/runtime#71146]() and [dotnet/runtime#71278](). While somewhat of a mouthful, these methods are quite handy. They do what their name suggests: whereas `IndexOf(T value)` searches for the first occurrence of `value` in the input, and whereas `IndexOfAny(T value0, T value1, ...)` searches for the first occurrence of any of `value0`, `value1`, etc. in the input, `IndexOfAnyExcept(T value)` searches for the first occurrence of something that's *not* equal to `value`, and similarly `IndexOfAnyExcept(T value0, T value1, ...)` searches for the first occurrence of something that's *not* equal to `value0`, `value1`, etc. For example, let's say you wanted to know whether an array of integers was entirely 0. You can now write that as:

```
bool allZero = array.AsSpan().IndexOfAnyExcept(0) < 0;
```

[dotnet/runtime#73488](https://github.com/dotnet/runtime/pull/73488) vectorizes this overload, as well.

```
private byte[] _zeros = new byte[1024];

[Benchmark(Baseline = true)]
public bool OpenCoded()
{
    foreach (byte b in _zeros)
    {
        if (b != 0)
        {
            return false;
        }
    }

    return true;
}

[Benchmark]
public bool IndexOfAnyExcept() => _zeros.AsSpan().IndexOfAnyExcept((byte)0) < 0;
```

Method	Mean	Ratio
OpenCoded	370.47 ns	1.00
IndexOfAnyExcept	23.84 ns	0.06

Of course, while new "index of" variations are helpful, we already have a bunch of such methods, and it's important that they are as efficient as possible. These core `IndexOf{Any}` methods are used in huge numbers of places, many of which are performance-sensitive, and so every release they get additional tender-loving care. While PRs like [dotnet/runtime#67811](https://github.com/dotnet/runtime/pull/67811) got gains by paying very close attention to the assembly code being generated (in this case, tweaking some of the checks used on Arm64 in `IndexOf` and `IndexOfAny` to achieve better utilization), the biggest improvements here come in places where either vectorization was added and none was previously employed, or where the vectorization scheme was overhauled for significant gain. Let's start with [dotnet/runtime#63285](https://github.com/dotnet/runtime/pull/63285), which yields huge improvements for many uses of `IndexOf` and `LastIndexOf` for "substrings" of `byte`s and `char`s. Previously, given a call like `str.IndexOf("hello")`, the implementation would essentially do the equivalent of repeatedly searching for the 'h', and when an 'h' was found, then performing a `SequenceEqual` to match the remainder. As you can imagine, however, it's very easy to run into cases where the first character being searched for is very common, such that you frequently have to break out of the vectorized loop in order to do the full string comparison. Instead, the PR implements an algorithm based on [SIMD-friendly algorithms for substring searching](http://0x80.pl/articles/simd-strfind.html). Rather than just searching for the first character, it can instead vectorize a search for both the first and last character at appropriate distances from each other. In our "hello" example, in any given input, it's much more likely to find an 'h' than it is to find an 'h' followed four characters later by an 'o', and thus this implementation is able to stay within the vectorized loop a lot longer, garnering many fewer false positives that force it down the `SequenceEqual` route. The implementation also handles cases where the two characters selected are equal, in which case it'll quickly look for another character that's not equal in order to maximize the efficiency of the search. We can see the impact of all of this with a couple of examples:

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;

[Benchmark]
[Arguments("Sherlock")]
[Arguments("elementary")]
public int Count(string needle)
{
    ReadOnlySpan<char> haystack = s_haystack;
    int count = 0, pos;
    while ((pos = haystack.IndexOf(needle)) >= 0)
    {
        haystack = haystack.Slice(pos + needle.Length);
        count++;
    }

    return count;
}
```

This is pulling down the text to "The Adventures of Sherlock Holmes" from Project Gutenberg and then benchmarking using `IndexOf` to count the occurrences of "Sherlock" and "elementary" in the text. On my machine, I get results like this:

Method	Runtime	needle	Mean	Ratio
Count	.NET 6.0	Sherlock	43.68 us	1.00
Count	.NET 7.0	Sherlock	48.33 us	1.11
Count	.NET 6.0	elementary	1,063.67 us	1.00
Count	.NET 7.0	elementary	56.04 us	0.05

For "Sherlock", the performance is actually a bit worse in .NET 7 than in .NET 6; not much, but a measurable 10%. That's because there are very few capital `'S'` characters in the source text, 841 to be exact, out of 593,836 characters in the document. At only 0.1% density of the starting character, the new algorithm doesn't bring much benefit, as the existing algorithm that searched for the first character alone captures pretty much all of the possible vectorization gains to be had, and we do pay a bit of overhead in doing a search for both the `'S'` and the `'k'`, whereas previously we'd have only searched for the `'S'`. In contrast, though, there are 54,614 `'e'` characters in the document, so almost 10% of the source. In that case, .NET 7 is 20x faster than .NET 6, taking 53us on .NET 7 to count all the `'e'`'s vs 1084us on .NET 6. In this case, the new scheme yields immense gains, by vectorizing a search for both the `'e'` and a `'y'` at the specific distance away, a combination that is much, much less frequent. This is one of those situations where overall there are on average huge observed gains even though we can see small regressions for some specific inputs.

Another example of significantly changing the algorithm employed is [dotnet/runtime#67758](https://github.com/dotnet/runtime/pull/67758), which enables some amount of vectorization to be applied to `IndexOf("...", StringComparison.OrdinalIgnoreCase)`. Previously, this operation was implemented with a fairly typical substring search, walking the input string and at every location doing an inner loop to compare the target string, except performing a `ToUpper` on every character in order to do it in a case-insensitive manner. Now with this PR, which is based on approaches previously used by `Regex`, if the target string begins with an ASCII character, the implementation can use `IndexOf` (if the character isn't an ASCII letter) or `IndexOfAny` (if the character is an ASCII letter) to quickly jump ahead to the first possible location of a match. Let's take the exact same benchmark as we just looked at, but tweaked to use `OrdinalIgnoreCase`:

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;

[Benchmark]
[Arguments("Sherlock")]
[Arguments("elementary")]
public int Count(string needle)
{
    ReadOnlySpan<char> haystack = s_haystack;
    int count = 0, pos;
    while ((pos = haystack.IndexOf(needle, StringComparison.OrdinalIgnoreCase)) >= 0)
    {
        haystack = haystack.Slice(pos + needle.Length);
        count++;
    }

    return count;
}
```

Here, both words are about 4x faster on .NET 7 than they were on .NET 6:

Method	Runtime	needle	Mean	Ratio
Count	.NET 6.0	Sherlock	2,113.1 us	1.00
Count	.NET 7.0	Sherlock	467.3 us	0.22
Count	.NET 6.0	elementary	2,325.6 us	1.00
Count	.NET 7.0	elementary	638.8 us	0.27

as we're now doing a vectorized `IndexOfAny('S', 's')` or `IndexOfAny('E', 'e')` rather than manually walking each character and comparing it. ([dotnet/runtime#73533](https://github.com/dotnet/runtime/pull/73533) uses the same approach now for handling `IndexOf(char, StringComparison.OrdinalIgnoreCase)`.)

Another example comes from [dotnet/runtime#67492](https://github.com/dotnet/runtime/pull/67492) from [@gfoidl](https://github.com/gfoidl). It updates `MemoryExtensions.Contains` with the approach we discussed earlier for handling the leftover elements at the end of vectorized operation: process one last vector's worth of data, even if it means duplicating some work already done. This particularly helps for smaller inputs where the processing time might otherwise be dominated by the serial handling of those leftovers.

```
private byte[] _data = new byte[95];
```

```
[Benchmark]
public bool Contains() => _data.AsSpan().Contains((byte)1);
```

Method	Runtime	Mean	Ratio
Contains	.NET 6.0	15.115 ns	1.00
Contains	.NET 7.0	2.557 ns	0.17

dotnet/runtime#60974 from [@alexcovington](https://github.com/alexcovington) broadens the impact of `IndexOf`. Prior to this PR, `IndexOf` was vectorized for one and two-byte sized primitive types, but this PR extends it as well to four and eight-byte sized primitives. As with most of the other vectorized implementations, it checks whether the `T` is bitwise-equatable, which is important for the vectorization as it's only looking at the bits in memory and not paying attention to any `Equals` implementation that might be defined on the type. In practice today, that means this is limited to just a handful of types of which the runtime has intimate knowledge (`Boolean`, `Byte`, `SByte`, `UInt16`, `Int16`, `Char`, `UInt32`, `Int32`, `UInt64`, `Int64`, `UIntPtr`, `IntPtr`, `Rune`, and enums), but in theory it could be extended in the future.

```
private int[] _data = new int[1000];

[Benchmark]
public int IndexOf() => _data.AsSpan().IndexOf(42);
```

Method	Runtime	Mean	Ratio
IndexOf	.NET 6.0	252.17 ns	1.00
IndexOf	.NET 7.0	78.82 ns	0.31

One final interesting `IndexOf`-related optimization. `string` has long had `IndexOf`/`IndexOfAny`/`LastIndexOf`/`LastIndexOfAny`, and obviously for `string` it's all about processing `char`s. When `ReadOnlySpan<T>` and `Span<T>` came on the scene, `MemoryExtensions` was added to provide extension methods for spans and friends, including such `IndexOf`/`IndexOfAny`/`LastIndexOf`/`LastIndexOfAny` methods. But for spans, this is about more than just `char`, and so `MemoryExtensions` grew its own set of implementations largely separate from `string`'s. Over the years, `MemoryExtensions` implementations have specialized more and more types, but in particular `byte` and `char`, such that over time `string`'s implementations have mostly been replaced by delegation into the same implementation as `MemoryExtensions` uses. However, `IndexOfAny` and `LastIndexOfAny` had been unification holdouts, each in its own direction. `string.IndexOfAny` did delegate to the same implementation as `MemoryExtensions.IndexOfAny` for 1-5 values being searched for, but for more than 5 values, `string.IndexOfAny` used a "probabilistic map," essentially a Bloom filter. It creates a 256-bit table, and quickly sets bits in that table based on the values being searched for (essentially hashing them, but with a trivial hash function). Then it iterates through the input, and rather than checking every input character against every one of the target values, it instead first looks up the input character in the table. If the corresponding bit isn't set, it knows the input character doesn't match any of the target values. If the corresponding bit is set, then it proceeds to compare the input character against each of the target values, with a high probability of it being one of them. `MemoryExtensions.IndexOfAny` lacked such a filter for more than 5 values. Conversely, `string.LastIndexOfAny` didn't provide any vectorization for multiple target values, whereas `MemoryExtensions.LastIndexOfAny` vectorized two and three target values. As of dotnet/runtime#63817, all of these are now unified, such that both `string` and `MemoryExtensions` get the best of what the other had.

```
private readonly char[] s_target = new[] { 'z', 'q' };
const string Sonnet = """
    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,
    And summer's lease hath all too short a date;
    Sometime too hot the eye of heaven shines,
    And often is his gold complexion dimm'd;
    And every fair from fair sometime declines,
    By chance or nature's changing course untrimm'd;
    But thy eternal summer shall not fade,
    Nor lose possession of that fair thou ow'st;
    Nor shall death brag thou wander'st in his shade,
    When in eternal lines to time thou grow'st:
    So long as men can breathe or eyes can see,
    So long lives this, and this gives life to thee.
    """;

[Benchmark]
public int LastIndexOfAny() => Sonnet.LastIndexOfAny(s_target);

[Benchmark]
public int CountLines()
{
    int count = 0;
    foreach (ReadOnlySpan<char> _ in Sonnet.AsSpan().EnumerateLines())
    {
        count++;
    }

    return count;
}
```

Method	Runtime	Mean	Ratio
LastIndexOfAny	.NET 6.0	443.29 ns	1.00
LastIndexOfAny	.NET 7.0	31.79 ns	0.07
CountLines	.NET 6.0	1,689.66 ns	1.00
CountLines	.NET 7.0	1,461.64 ns	0.86

That same PR also cleans up uses of the `IndexOf` family, and in particular around uses that are checking for containment rather than the actual index of a result. The `IndexOf` family of methods return a non-negative value when an element is found, and otherwise return -1. That means when checking whether an element was found, code can use either `>= 0` or `!= -1`, and when checking whether an element wasn't found, code can use either `< 0` or `== -1`. It turns out that the code generated for comparisons against 0 is ever so slightly more efficient than comparisons generated against -1, and this isn't something the JIT can itself substitute without the `IndexOf` methods being intrinsics such that the JIT can understand the semantics of the return value. Thus, for consistency and a small perf gain, all relevant call sites were switched to compare against 0 instead of against -1.

Speaking of call sites, one of the great things about having highly optimized `IndexOf` methods is using them in all the places that can benefit, removing the maintenance impact of open-coded replacements while also reaping the perf wins. [dotnet/runtime#63913](https://github.com/dotnet/runtime/pull/63913) used `IndexOf` inside of `StringBuilder.Replace` to speed up the search for the next character to be replaced:

```
private StringBuilder _builder = new StringBuilder(Sonnet);

[Benchmark]
public void Replace()
{
    _builder.Replace('?', '!');
    _builder.Replace('!', '?');
}
```

Method	Runtime	Mean	Ratio
Replace	.NET 6.0	1,563.69 ns	1.00
Replace	.NET 7.0	70.84 ns	0.04

[dotnet/runtime#60463](https://github.com/dotnet/runtime/pull/60463) from [@nietras](https://github.com/nietras) used `IndexOfAny` in `StringReader.ReadLine` to search for `'\r'` and `'\n'` line ending characters, which results in some substantial throughput gains even with the allocation and copy that is inherent to the method's design:

```
[Benchmark]
public void ReadAllLines()
{
    var reader = new StringReader(Sonnet);
    while (reader.ReadLine() != null) ;
}
```

Method	Runtime	Mean	Ratio
ReadAllLines	.NET 6.0	947.8 ns	1.00
ReadAllLines	.NET 7.0	385.7 ns	0.41

And [dotnet/runtime#70176](https://github.com/dotnet/runtime/pull/70176) cleaned up a plethora of additional uses.

Finally on the `IndexOf` front, as noted, a lot of time and energy over the years has gone into optimizing these methods. In previous releases, some of that energy has been in the form of using hardware intrinsics directly, e.g. having an SSE2 code path and an AVX2 code path and an AdvSimd code path. Now that we have `Vector128<T>` and `Vector256<T>`, many such uses can be simplified (e.g. avoiding the duplication between an SSE2 implementation and an AdvSimd implementation) while still maintaining as good or even better performance and while automatically supporting vectorization on other platforms with their own intrinsics, like WebAssembly. [dotnet/runtime#73481](https://github.com/dotnet/runtime/pull/73481), [dotnet/runtime#73556](https://github.com/dotnet/runtime/pull/73556), [dotnet/runtime#73368](https://github.com/dotnet/runtime/pull/73368), [dotnet/runtime#73364](https://github.com/dotnet/runtime/pull/73364), [dotnet/runtime#73064](https://github.com/dotnet/runtime/pull/73064), and [dotnet/runtime#73469](https://github.com/dotnet/runtime/pull/73469) all contributed here, in some cases incurring meaningful throughput gains:

```
[Benchmark]
public int IndexOfAny() => Sonnet.AsSpan().IndexOfAny("!.<>");
```

Method	Runtime	Mean	Ratio
IndexOfAny	.NET 6.0	52.29 ns	1.00
IndexOfAny	.NET 7.0	40.17 ns	0.77

The `IndexOf` family is just one of many on `string`/`MemoryExtensions` that has seen dramatic improvements. Another are the `SequenceEquals` family, including `Equals`, `StartsWith`, and `EndsWith`. One of my favorite changes in the whole release is [dotnet/runtime#65288](https://github.com/dotnet/runtime/pull/65288) and is squarely in this area. It's very common to see calls to methods like `StartsWith` with a constant string argument, e.g. `value.StartsWith("https://")`, `value.SequenceEquals("Key")`, etc. These methods are now recognized by the JIT, which can now automatically unroll the comparison and compare more than one char at a time, e.g. doing a single read of four chars as a `long` and a single comparison of that `long` against the expected combination of those four chars. The result is beautiful. Making it even better is [dotnet/runtime#66095](https://github.com/dotnet/runtime/pull/66095), which adds to this support for `OrdinalIgnoreCase`. Remember those ASCII bit twiddling tricks discussed a bit earlier with `char.IsAsciiLetter` and friends? The JIT now employs the same trick as part of this unrolling, so if you do that same `value.StartsWith("https://")` but instead as `value.StartsWith("https://", StringComparison.OrdinalIgnoreCase)`, it will recognize that the whole comparison string is ASCII and will OR in the appropriate mask on both the comparison constant and on the read data from the input in order to perform the comparison in a case-insensitive manner.

```
private string _value = "https://dot.net";

[Benchmark]
public bool IsHttps_Ordinal() => _value.StartsWith("https://", StringComparison.Ordinal);

[Benchmark]
public bool IsHttps_OrdinalIgnoreCase() => _value.StartsWith("https://",
StringComparison.OrdinalIgnoreCase);
```

Method	Runtime	Mean	Ratio
IsHttps_Ordinal	.NET 6.0	4.5634 ns	1.00
IsHttps_Ordinal	.NET 7.0	0.4873 ns	0.11
IsHttps_OrdinalIgnoreCase	.NET 6.0	6.5654 ns	1.00
IsHttps_OrdinalIgnoreCase	.NET 7.0	0.5577 ns	0.08

Interestingly, since .NET 5 the code generated by `RegexOptions.Compiled` would perform similar unrolling when comparing sequences of multiple characters, and when the source generator was added in .NET 7, it also learned how to do this. However, the source generator has problems with such an optimization, due to endianness. The constants being compared against are subject to byte ordering issues, such that the source generator would need to emit code that could handle running on either little-endian or big-endian machines. The JIT has no such problem, as it's generating the code on the same machine on which the code will execute (and in scenarios where it's being used to generate code ahead of time, the entirety of that code is already tied to a particular architecture). By moving this optimization into the JIT, the corresponding code could be deleted from `RegexOptions.Compiled` and the regex source generator, which then also benefits from producing much easier to read code utilizing `StartsWith` that's just as fast ([dotnet/runtime#65222](https://github.com/dotnet/runtime/pull/65222) and [dotnet/runtime#66339](https://github.com/dotnet/runtime/pull/66339)). Wins all around. (This could only be removed from `RegexOptions.Compiled` after [dotnet/runtime#68055](https://github.com/dotnet/runtime/pull/68055), which fixed the ability for the JIT to recognize these string literals in `DynamicMethods`, which `RegexOptions.Compiled` uses with reflection emit to spit out the IL for the regex being compiled.)

`StartsWith` and `EndsWith` have improved in other ways. [dotnet/runtime#63734](https://github.com/dotnet/runtime/pull/63734) (improved further by [dotnet/runtime#64530](https://github.com/dotnet/runtime/pull/64530)) added another really interesting JIT-based optimization, but to understand it, we need to understand `string`'s internal layout. `string` is essentially represented in memory as an `int` length followed by that many `char`s plus a null terminator `char`. The actual `System.String` class represents this in C# as an `int _stringLength` field followed by a `char _firstChar` field, such that `_firstChar` indeed lines up with the first character of the string, or the null terminator if the string is empty. Internally in System.Private.CoreLib, and in particular in methods on `string` itself, code will often refer to `_firstChar` directly when the first character needs to be consulted, as it's typically faster to do that than to use `str[0]`, in particular because there are no bounds checks involved and the string's length generally needn't be consulted. Now, consider a method like `public bool StartsWith(char value)` on `string`. In .NET 6, the implementation was:

```
return Length != 0 && _firstChar == value;
```

which given what I just described makes sense: if the `Length` is 0, then the string doesn't begin with the specified character, and if `Length` is not 0, then we can just compare the value against `_firstChar`. But, why is that `Length` check even needed at all? Couldn't we just do `return _firstChar == value;`? That will avoid the additional comparison and branch, and it will work just fine… unless the target character is itself `'\0'`, in which case we could get false positives on the result. Now to this PR. The PR introduces an internal JIT intrinsinc `RuntimeHelpers.IsKnownConstant`, which the JIT will substitute with `true` if the containing method is inlined and the argument passed to `IsKnownConstant` is then seen to be a constant. In such cases, the implementation can rely on other JIT optimizations kicking in and optimizing various code in the method, effectively enabling a developer to write two different implementations, one when the argument is known to be a constant and one when not. With that in hand, the PR is able to optimize `StartsWith` as follows:

```
public bool StartsWith(char value)
{
    if (RuntimeHelpers.IsKnownConstant(value) && value != '\0')
        return _firstChar == value;

    return Length != 0 && _firstChar == value;
}
```

If the `value` parameter isn't a constant, then `IsKnownConstant` will be substituted with `false`, the entire starting `if` block will be eliminated, and the method will be left exactly was it was before. But, if this method gets inlined and the `value` was actually a constant, then the `value != '\0'` condition will also be evaluatable at JIT-compile-time. If the `value` is in fact `'\0'`, well, again that whole `if` block will be eliminated and we're no worse off. But in the common case where the `value` isn't null, the entire method will end up being compiled as if it were:

```
return _firstChar == ConstantValue;
```

and we've saved ourselves a read of the string's length, a comparison, and a branch. dotnet/runtime#69038 then employs a similar technique for `EndsWith`.

```
private string _value = "https://dot.net";

[Benchmark]
public bool StartsWith() =>
    _value.StartsWith('a') ||
    _value.StartsWith('b') ||
    _value.StartsWith('c') ||
    _value.StartsWith('d') ||
    _value.StartsWith('e') ||
    _value.StartsWith('f') ||
    _value.StartsWith('g') ||
    _value.StartsWith('i') ||
    _value.StartsWith('j') ||
    _value.StartsWith('k') ||
    _value.StartsWith('l') ||
    _value.StartsWith('m') ||
    _value.StartsWith('n') ||
    _value.StartsWith('o') ||
    _value.StartsWith('p');
```

Method	Runtime	Mean	Ratio
StartsWith	.NET 6.0	8.130 ns	1.00
StartsWith	.NET 7.0	1.653 ns	0.20

(Another example of `IsKnownConstant` being used comes from dotnet/runtime#64016, which uses it to improve `Math.Round` when a `MidpointRounding` mode is specified. Call sites to this almost always explicitly specify the enum value as a constant, which then allows the JIT to specialize the code generation for the method to the specific mode being used; that in turn, for example, enables a `Math.Round(..., MidpointRounding.AwayFromZero)` call on Arm64 to be lowered to a single `frinta` instruction.)

`EndsWith` was also improved in dotnet/runtime#72750, and specifically for when `StringComparison.OrdinalIgnoreCase` is specified. This simple PR just switched which internal helper method was used to implement this method, taking advantage of one that is sufficient for the needs of this method and that has lower overheads.

```
[Benchmark]
[Arguments("System.Private.CoreLib.dll", ".DLL")]
public bool EndsWith(string haystack, string needle) =>
    haystack.EndsWith(needle, StringComparison.OrdinalIgnoreCase);
```

Method	Runtime	Mean	Ratio
EndsWith	.NET 6.0	10.861 ns	1.00
EndsWith	.NET 7.0	5.385 ns	0.50

Finally, dotnet/runtime#67202 and dotnet/runtime#73475 employ `Vector128<T>` and `Vector256<T>` to replace direct hardware intrinsics usage, just as was previously shown for various `IndexOf` methods, but here for `SequenceEqual` and `SequenceCompareTo`, respectively.

Another method that's seem some attention in .NET 7 is `MemoryExtensions.Reverse` (and `Array.Reverse` as it shares the same implementation), which performs an in-place reversal of the target span. [dotnet/runtime#64412](https://github.com/dotnet/runtime/pull/64412) from [@alexcovington](https://github.com/alexcovington) provides a vectorized implementation via direct use of AVX2 and SSSE3 hardware intrinsics, with [dotnet/runtime#72780](https://github.com/dotnet/runtime/pull/72780) from [@SwapnilGaikwad](https://github.com/SwapnilGaikwad) following up to add an AdvSimd intrinsics implementation for Arm64. (There was an unintended regression introduced by the original vectorization change, but that was fixed by [dotnet/runtime#70650](https://github.com/dotnet/runtime/pull/70650).)

```
private char[] text = "Free. Cross-platform. Open source.\r\nA developer platform for
building all your apps.".ToCharArray();

[Benchmark]
public void Reverse() => Array.Reverse(text);
```

Method	Runtime	Mean	Ratio
Reverse	.NET 6.0	21.352 ns	1.00
Reverse	.NET 7.0	9.536 ns	0.45

`String.Split` also saw vectorization improvements in [dotnet/runtime#64899](https://github.com/dotnet/runtime/pull/64899) from [@yesmey](https://github.com/yesmey). As with some of the previously discussed PRs, it switched the existing usage of SSE2 and SSSE3 hardware intrinsics over to the new `Vector128<T>` helpers, which improved upon the existing implementation while also implicitly adding vectorization support for Arm64.

Converting various formats of strings is something many applications and services do, whether that's converting from UTF8 bytes to and from `string` or formatting and parsing hex values. Such operations have also improved in a variety of ways in .NET 7. [Base64-encoding](https://en.wikipedia.org/wiki/Base64), for example, is a way of representing arbitrary binary data (think `byte[]`) across mediums that only support text, encoding bytes into one of 64 different ASCII characters. Multiple APIs in .NET implement this encoding. For converting between binary data represented as `ReadOnlySpan<byte>` and UTF8 (actually ASCII) encoded data also represented as `ReadOnlySpan<byte>`, the `System.Buffers.Text.Base64` type provides `EncodeToUtf8` and `DecodeFromUtf8` methods. These were vectorized several releases ago, but they were further improved in .NET 7 via [dotnet/runtime#70654](https://github.com/dotnet/runtime/pull/70654) from [@a74nh](https://github.com/a74nh), which converted the SSSE3-based implementation to use `Vector128<T>` (which in turn implicitly enabled vectorization on Arm64). However, for converting between arbitrary binary data represented as `ReadOnlySpan<byte>`/`byte[]` and `ReadOnlySpan<char>`/`char[]`/`string`, the `System.Convert` type exposes multiple methods, e.g. `Convert.ToBase64String`, and these methods historically were not vectorized. That changes in .NET 7, where [dotnet/runtime#71795](https://github.com/dotnet/runtime/pull/71795) and [dotnet/runtime#73320](https://github.com/dotnet/runtime/pull/73320) vectorize the `ToBase64String`, `ToBase64CharArray`, and `TryToBase64Chars` methods. The way they do this is interesting. Rather than effectively duplicating the vectorization implementation from `Base64.EncodeToUtf8`, they instead layer on top of `EncodeToUtf8`, calling it to encode the input `byte` data into an output `Span<byte>`. Then, then they "widen" those bytes into chars (remember, Base64-encoded data is a set of ASCII chars, so going from these bytes to chars entails adding just a `0` byte onto each element). That widening can itself easily be done in a vectorized manner. The other interesting thing about this layering is it doesn't actually require separate intermediate storage for the encoded bytes. The implementation can perfectly compute the number of resulting characters for encoding X bytes into Y

Base64 characters (there's a formula), and the implementation can either allocate that final space (e.g. in the case of ToBase64CharArray) or ensure the provided space is sufficient (e.g. in the case of TryToBase64Chars). And since we know the initial encoding will require exactly half as many bytes, we can encode into that same space (with the destination span reinterpreted as a `byte` span rather than `char` span), and then widen "in place": walk from the end of the bytes and the end of the char space, copying the bytes into the destination.

```
private byte[] _data = Encoding.UTF8.GetBytes("""
    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,
    And summer's lease hath all too short a date;
    Sometime too hot the eye of heaven shines,
    And often is his gold complexion dimm'd;
    And every fair from fair sometime declines,
    By chance or nature's changing course untrimm'd;
    But thy eternal summer shall not fade,
    Nor lose possession of that fair thou ow'st;
    Nor shall death brag thou wander'st in his shade,
    When in eternal lines to time thou grow'st:
    So long as men can breathe or eyes can see,
    So long lives this, and this gives life to thee.
    """);
private char[] _encoded = new char[1000];

[Benchmark]
public bool TryToBase64Chars() => Convert.TryToBase64Chars(_data, _encoded, out _);
```

Method	Runtime	Mean	Ratio
TryToBase64Chars	.NET 6.0	623.25 ns	1.00
TryToBase64Chars	.NET 7.0	81.82 ns	0.13

Just as widening can be used to go from bytes to chars, narrowing can be used to go from chars to bytes, in particular if the chars are actually ASCII and thus have a 0 upper byte. Such narrowing can be vectorized, and the internal `NarrowUtf16ToAscii` utility helper does exactly that, used as part of methods like `Encoding.ASCII.GetBytes`. While this method was previously vectorized, its primary fast-path utilized SSE2 and thus didn't apply to Arm64; thanks to [dotnet/runtime#70080](https://github.com/dotnet/runtime/pull/70080) from [@SwapnilGaikwad](https://github.com/SwapnilGaikwad), that path was changed over to be based on the cross-platform `Vector128<T>`, enabling the same level of optimization across supported platforms. Similarly, [dotnet/runtime#71637](https://github.com/dotnet/runtime/pull/71637) from [@SwapnilGaikwad](https://github.com/SwapnilGaikwad) adds Arm64 vectorization to the `GetIndexOfFirstNonAsciiChar` internal helper that's used by methods like `Encoding.UTF8.GetByteCount`. (And in the same vein, [dotnet/runtime#67192](https://github.com/dotnet/runtime/pull/67192) changed the internal `HexConverter.EncodeToUtf16` method from using SSSE3 intrinsics to instead use `Vector128<T>`, automatically providing an Arm64 implementation.)

`Encoding.UTF8` was also improved a bit. In particular, [dotnet/runtime#69910](https://github.com/dotnet/runtime/pull/69910) streamlined the implementations of `GetMaxByteCount` and `GetMaxCharCount`, making them small enough to be commonly inlined when used directly off of `Encoding.UTF8` such that the JIT is able to devirtualize the calls.

```
[Benchmark]
public int GetMaxByteCount() => Encoding.UTF8.GetMaxByteCount(Sonnet.Length);
```

Method	Runtime	Mean	Ratio
GetMaxByteCount	.NET 6.0	1.7442 ns	1.00
GetMaxByteCount	.NET 7.0	0.4746 ns	0.27

Arguably the biggest improvement around UTF8 in .NET 7 is the new C# 11 support for UTF8 literals. Initially implemented in the C# compiler in [dotnet/roslyn#58991](), with follow-on work in [dotnet/roslyn#59390](), [dotnet/roslyn#61532](), and [dotnet/roslyn#62044](), UTF8 literals enables the compiler to perform the UTF8 encoding into bytes at compile-time. Rather than writing a normal string, e.g. `"hello"`, a developer simply appends the new `u8` suffix onto the string literal, e.g. `"hello"u8`. At that point, this is no longer a string. Rather, the natural type of this expression is a `ReadOnlySpan<byte>`. If you write:

```
public static ReadOnlySpan<byte> Text => "hello"u8;
```

the C# compiler will compile that equivalent to if you wrote:

```
public static ReadOnlySpan<byte> Text =>
    new ReadOnlySpan<byte>(new byte[] { (byte)'h', (byte)'e', (byte)'l', (byte)'l',
(byte)'o', (byte)'\0' }, 0, 5);
```

In other words, the compiler is doing the equivalent of `Encoding.UTF8.GetBytes` at compile-time and hardcoding the resulting bytes, saving the cost of performing that encoding at run-time. Of course, at first glance, that array allocation might look terribly inefficient. However, looks can be deceiving, and are in this case. For several releases now, when the C# compiler sees a `byte[]` (or `sbyte[]` or `bool[]`) being initialized with a constant length and constant values and immediately cast to or used to construct a `ReadOnlySpan<byte>`, it optimizes away the `byte[]` allocation. Instead, it blits the data for that span into the assembly's data section, and then constructs a span that points directly to that data in the loaded assembly. This is the actual generated IL for the above property:

```
IL_0000: ldsflda valuetype '<PrivateImplementationDetails>'/'__StaticArrayInitTypeSize=6'
'<PrivateImplementationDetails>'::F3AEFE62965A91903610F0E23CC8A69D5B87CEA6D28E75489B0D2CA02
ED7993C
IL_0005: ldc.i4.5
IL_0006: newobj instance void valuetype
[System.Runtime]System.ReadOnlySpan`1<uint8>::.ctor(void*, int32)
IL_000b: ret
```

This means we not only save on the encoding costs at run-time, and we not only avoid whatever managed allocations might be required to store the resulting data, we also benefit from the JIT being able to see information about the encoded data, like it's length, enabling knock-on optimizations. You can see this clearly by examining the assembly generated for a method like:

```
public static int M() => Text.Length;
```

for which the JIT produces:

```
; Program.M()
       mov       eax,5
```

```
        ret
; Total bytes of code 6
```

The JIT inlines the property access, sees that the span is being constructed with a length of `5`, and so rather than emitting any array allocations or span constructions or anything even resembling that, it simply outputs `mov eax, 5` to return the known length of the span.

Thanks primarily to [dotnet/runtime#70568](), [dotnet/runtime#69995](), [dotnet/runtime#70894](), [dotnet/runtime#71417]() from [@am11](https://github.com/am11), [dotnet/runtime#71292](), [dotnet/runtime#70513](), and [dotnet/runtime#71992](), `u8` is now used more than 2100 times throughout [dotnet/runtime](). Hardly a fair comparison, but the following benchmark demonstrates how little work is actually being performed for `u8` at execution time:

```
[Benchmark(Baseline = true)]
public ReadOnlySpan<byte> WithEncoding() => Encoding.UTF8.GetBytes("test");

[Benchmark]
public ReadOnlySpan<byte> Withu8() => "test"u8;
```

Method	Mean	Ratio	Allocated	Alloc Ratio
WithEncoding	17.3347 ns	1.000	32 B	1.00
Withu8	0.0060 ns	0.000	-	0.00

Like I said, not fair, but it proves the point :)

`Encoding` is of course just one mechanism for creating `string` instances. Others have also improved in .NET 7. Take the super common `long.ToString`, for example. Previous releases improved `int.ToString`, but there were enough differences between the 32-bit and 64-bit algorithms that `long` didn't see all of the same gains. Now thanks to [dotnet/runtime#68795](), the 64-bit formatting code paths are made much more similar to the 32-bit, resulting in faster performance.

You can also see improvements in `string.Format` and `StringBuilder.AppendFormat`, as well as other helpers that layer on top of these (like `TextWriter.AppendFormat`). [dotnet/runtime#69757]() overhauls the core routines inside `Format` to avoid unnecessary bounds checking, favor expected cases, and generally clean up the implementation. It also, however, utilities `IndexOfAny` to search for the next interpolation hole that needs to be filled in, and if the non-hole-character to hole ratio is high (e.g. long format string with few holes), it can be way faster than before.

```
private StringBuilder _sb = new StringBuilder();

[Benchmark]
public void AppendFormat()
{
    _sb.Clear();
    _sb.AppendFormat("There is already one outstanding '{0}' call for this WebSocket
instance." +
                     "ReceiveAsync and SendAsync can be called simultaneously, but at most
one " +
                     "outstanding operation for each of them is allowed at the same time.",
                     "ReceiveAsync");
}
```

Method	Runtime	Mean	Ratio
AppendFormat	.NET 6.0	338.23 ns	1.00
AppendFormat	.NET 7.0	49.15 ns	0.15

Speaking of `StringBuilder`, it's seen additional improvements beyond the aforementioned changes to `AppendFormat`. One interesting change is [dotnet/runtime#64405](), which achieved two related things. The first was to remove pinning as part of formatting operations. As an example, `StringBuilder` has an `Append(char* value, int valueCount)` overload which copies the specified number of characters from the specified pointer into the `StringBuilder`, and other APIs were implemented in terms of this method; for example, the `Append(string? value, int startIndex, int count)` method was essentially implemented as:

```
fixed (char* ptr = value)
{
    Append(ptr + startIndex, count);
}
```

That `fixed` statement translates into a "pinning pointer." Normally the GC is free to move managed objects around on the heap, which it might do in order to compact the heap (to, for example, avoid small, unusuable fragments of memory between objects). But if the GC can move objects around, a normal native pointer into that memory would be terribly unsafe and unreliable, as without notice the data being pointed to could move and your pointer could now be pointing to garbage or to some other object that was shifted to this location. There are two ways for dealing with this. The first is a "managed pointer," otherwise known as a "reference" or "ref," as that's exactly what you get when you have the "ref" keyword in C#; it's a pointer that the runtime will update with the correct value when it moves the object being pointed into. The second is to prevent the pointed-to object from being moved, "pinning" it in place. And that's what the "fixed" keyword does, pinning the referenced object for the duration of the `fixed` block, during which time it's safe to use the supplied pointer. Thankfully, pinning is cheap when no GC occurs; when a GC does occur, however, pinned objects aren't able to be moved around, and thus pinning can have a global impact on the performance of the application (and on GCs themselves). There are also various optimizations inhibited by pinning. With all of the advents in C# around being able to use `ref` in many more places (e.g. ref locals, ref returns, and now in C# 11, ref fields), and with all of the new APIs in .NET for manipulating refs (e.g. `Unsafe.Add`, `Unsafe.AreSame`), it's now possible to rewrite code that was using pinning pointers to instead use managed pointers, thereby avoiding the problems that come from pinning. Which is what this PR did. Rather than implementing all of the `Append` methods in terms of an `Append(char*, int)` helper, they're now all implemented in terms of an `Append(ref char, int)` helper. So for example instead of the previously shown `Append(string? value, int startIndex, int count)` implementation, it's now akin to

```
Append(ref Unsafe.Add(ref value.GetRawStringData(), startIndex), count);
```

where that `string.GetRawStringData` method is just an internal version of the public `string.GetPinnableReference` method, returning a `ref` instead of a `ref readonly`. This means that all of the high-performance code inside of `StringBuilder` that had been using pointers to avoid bounds checking and the like can continue to do so, but now also does so without pinning all of the inputs.

The second thing this `StringBuilder` change did was unify an optimization that was present for `string` inputs to also apply to `char[]` inputs and `ReadOnlySpan<char>` inputs. Specifically, because it's so common to append `string` instances to a `StringBuilder`, a special code path was long ago put in place to optimize for this input and specifically for the case where there's already enough room in the `StringBuilder` to hold the whole input, at which point an efficient copy can be used. With a shared `Append(ref char, int)` helper, though, this optimization can be moved down into that helper, such that it not only helps out `string` but any other type that also calls into the same helper. The effects of this are visible in a simple microbenchmark:

```
private StringBuilder _sb = new StringBuilder();

[Benchmark]
public void AppendSpan()
{
    _sb.Clear();
    _sb.Append("this".AsSpan());
    _sb.Append("is".AsSpan());
    _sb.Append("a".AsSpan());
    _sb.Append("test".AsSpan());
    _sb.Append(".".AsSpan());
}
```

Method	Runtime	Mean	Ratio
AppendSpan	.NET 6.0	35.98 ns	1.00
AppendSpan	.NET 7.0	17.59 ns	0.49

One of the great things about improving things low in the stack is they have a multiplicative effect; they not only help improve the performance of user code that directly relies on the improved functionality, they can also help improve the performance of other code in the core libraries, which then further helps dependent apps and services. You can see this, for example, with `DateTimeOffset.ToString`, which depends on `StringBuilder`:

```
private DateTimeOffset _dto = DateTimeOffset.UtcNow;

[Benchmark]
public string DateTimeOffsetToString() => _dto.ToString();
```

Method	Runtime	Mean	Ratio
DateTimeOffsetToString	.NET 6.0	340.4 ns	1.00
DateTimeOffsetToString	.NET 7.0	289.4 ns	0.85

`StringBuilder` itself was then further updated by [dotnet/runtime#64922](https://github.com/dotnet/runtime/pull/64922) from [@teo-tsirpanis](https://github.com/teo-tsirpanis), which improves the `Insert` methods. It used to be that the `Append(primitive)` methods on `StringBuilder` (e.g. `Append(int)`) would call `ToString` on the value and then append the resulting string. With the advent of `ISpanFormattable`, as a fast-path those methods now try to format the value directly into the `StringBuilder`'s internal buffer, and only if there's not enough room remaining do they then take the old path as a fallback. `Insert` wasn't improved in this way at the time, because it can't just format into the space at the end of the builder; the insert location could be anywhere in the builder. This PR addresses that by formatting into some temporary stack space, and then delegating to the existing internal ref-based helper from the previously discussed PR to insert the resulting characters at the right location (it also falls back to `ToString` when there's not enough stack space for the `ISpanFormattable.TryFormat`, but that only happens in incredibly corner cases, like a floating-point value that formats to hundreds of digits).

```
private StringBuilder _sb = new StringBuilder();

[Benchmark]
public void Insert()
{
    _sb.Clear();
    _sb.Insert(0, 12345);
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Insert	.NET 6.0	30.02 ns	1.00	32 B	1.00
Insert	.NET 7.0	25.53 ns	0.85	-	0.00

Other minor improvements to `StringBuilder` have also been made, like [dotnet/runtime#60406](https://github.com/dotnet/runtime/pull/60406) which removed a small `int[]` allocation from the `Replace` method. Even with all these improvements, though, the fastest use of `StringBuilder` is no use; [dotnet/runtime#68768](https://github.com/dotnet/runtime/pull/68768) removed a bunch of uses of `StringBuilder` that would have been better served with other `string`-creation mechanisms. For example, the legacy `DataView` type had some code that created a sorting specification as a string:

```
private static string CreateSortString(PropertyDescriptor property, ListSortDirection
direction)
{
    var resultString = new StringBuilder();
    resultString.Append('[');
    resultString.Append(property.Name);
    resultString.Append(']');
    if (ListSortDirection.Descending == direction)
    {
        resultString.Append(" DESC");
    }
    return resultString.ToString();
}
```

We don't actually need the `StringBuilder` here, as in the worst-case we're just concatenating three strings, and `string.Concat` has a dedicated overload for that exact operation that has the best possible implementation for that operation (and if we ever found a better way, that method would be improved according). So we can just use that:

```
private static string CreateSortString(PropertyDescriptor property, ListSortDirection
direction) =>
    direction == ListSortDirection.Descending ?
        $"[{property.Name}] DESC" :
        $"[{property.Name}]";
```

Note that I've expressed that concatenation via an interpolated string, but the C# compiler will "lower" this interpolated string to a call to `string.Concat`, so the IL for this is indistinguishable from if I'd instead written:

```
private static string CreateSortString(PropertyDescriptor property, ListSortDirection
direction) =>
```

```
    direction == ListSortDirection.Descending ?
        string.Concat("[", property.Name, "] DESC") :
        string.Concat("[", property.Name, "]");
```

As an aside, the expanded `string.Concat` version highlights that this method could have been written to result in a bit less IL if it were instead written as:

```
private static string CreateSortString(PropertyDescriptor property, ListSortDirection
direction) =>
    string.Concat("[", property.Name, direction == ListSortDirection.Descending ? "] DESC"
: "]");
```

but this doesn't meaningfully affect performance and here clarity and maintainability was more important than shaving off a few bytes.

```
[Benchmark(Baseline = true)]
[Arguments("SomeProperty", ListSortDirection.Descending)]
public string WithStringBuilder(string name, ListSortDirection direction)
{
    var resultString = new StringBuilder();
    resultString.Append('[');
    resultString.Append(name);
    resultString.Append(']');
    if (ListSortDirection.Descending == direction)
    {
        resultString.Append(" DESC");
    }
    return resultString.ToString();
}

[Benchmark]
[Arguments("SomeProperty", ListSortDirection.Descending)]
public string WithConcat(string name, ListSortDirection direction) =>
    direction == ListSortDirection.Descending?
        $"[{name}] DESC" :
        $"[{name}]";
```

Method	Mean	Ratio	Allocated	Alloc Ratio
WithStringBuilder	68.34 ns	1.00	272 B	1.00
WithConcat	20.78 ns	0.31	64 B	0.24

There are also places where `StringBuilder` was still applicable, but it was being used on hot-enough paths that previous releases of .NET saw the `StringBuilder` instance being cached. Several of the core libraries, including System.Private.CoreLib, have an internal `StringBuilderCache` type which caches a `StringBuilder` instance in a `[ThreadStatic]`, meaning every thread could end up having such an instance. There are several issues with this, including that the buffers employed by `StringBuilder` aren't usable for anything else while the `StringBuilder` isn't in use, and because of that, `StringBuilderCache` places a limit on the capacity of the `StringBuilder` instances that can be cached; attempts to cache ones longer than that result in them being thrown away. It'd be better instead to use cached arrays that aren't length-limited and that everyone has access to for sharing. Many of the core .NET libraries have an internal `ValueStringBuilder` type for this purpose, a `ref struct`-based type that can use `stackalloc`'d memory to start and then if necessary grow into `ArrayPool<char>` arrays. And with dotnet/runtime#64522 and dotnet/runtime#69683, many of the remaining uses of `StringBuilderCache` have been replaced. I'm hopeful we can entirely remove `StringBuilderCache` in the future.

In the same vein of not doing unnecessary work, there's a fairly common pattern that shows up with methods like `string.Substring` and `span.Slice`:

```
span = span.Slice(offset, str.Length - offset);
```

The relevant thing to recognize here is these methods have overloads that take just the starting offset. Since the length being specified is the remainder after the specified offset, the call could instead be simplified to:

```
span = span.Slice(offset);
```

which is not only more readable and maintainable, it has some small efficiency benefits, e.g. on 64-bit the `Slice(int, int)` constructor has an extra addition over `Slice(int)`, and for 32-bit the `Slice(int, int)` constructor incurs an additional comparison and branch. It's thus beneficial for both code maintenance and for performance to simplify these calls, which dotnet/runtime#68937 does for all found occurrences of that pattern. This is then made more impactful by dotnet/runtime#73882, which streamlines `string.Substring` to remove unnecessary overheads, e.g. it condenses four argument validation checks down to a single fast-path comparison (in 64-bit processes).

Ok, enough about `string`. What about spans? One of the coolest features in C# 11 is the new support for `ref` fields. What is a `ref` field? You're familiar with `ref`s in C# in general, and we've already discussed how they're essentially managed pointers, i.e. pointers that the runtime can update at any time due to the object it references getting moved on the heap. These references can point to the beginning of an object, or they can point somewhere inside the object, in which case they're referred to as "interior pointers." `ref` has existed in C# since 1.0, but at that time it was primarily about passing by reference to method calls, e.g.

```
class Data
{
    public int Value;
}
...
void Add(ref int i)
{
    i++;
}
...
var d = new Data { Value = 42 };
Add(ref d.Value);
Debug.Assert(d.Value == 43);
```

Later versions of C# added the ability to have local `ref`s, e.g.

```
void Add(ref int i)
{
    ref j = ref i;
    j++;
}
```

and even to have `ref` returns, e.g.

```
ref int Add(ref int i)
{
    ref j = ref i;
    j++;
    return ref j;
}
```

These facilities are more advanced, but they're used liberally throughout higher-performance code bases, and many of the optimizations in .NET in recent years are possible in large part due to these `ref`-related capabilities.

`Span<T>` and `ReadOnlySpan<T>` themselves are heavily-based on `ref`s. For example, the indexer on many older collection types is implemented as a get/set property, e.g.

```
private T[] _items;
...
public T this[int i]
{
    get => _items[i];
    set => _items[i] = value;
}
```

But not span. `Span<T>`'s indexer looks more like this:

```
public ref T this[int index]
{
    get
    {
        if ((uint)index >= (uint)_length)
            ThrowHelper.ThrowIndexOutOfRangeException();

        return ref Unsafe.Add(ref _reference, index);
    }
}
```

Note there's only a getter and no setter; that's because it returns a `ref T` to the actual storage location. It's a writable ref, so you can assign to it, e.g. you can write:

```
span[i] = value;
```

but rather than that being equivalent to calling some setter:

```
span.set_Item(i, value);
```

it's actually equivalent to using the getter to retrieve the `ref` and then writing a value through that `ref`, e.g.

```
ref T item = ref span.get_Item(i);
item = value;
```

That's all well and good, but what's that `_reference` in the getter definition? Well, `Span<T>` is really just a tuple of two fields: a reference (to the start of the memory being referred to) and a length (how many elements from that reference are included in the span). In the past, the runtime had to hack this with an internal type (`ByReference<T>`) specially recognized by the runtime to be a reference. But as of C# 11 and .NET 7, `ref structs` can now contain `ref` fields, which means `Span<T>` today is literally defined as follows:

```csharp
public readonly ref struct Span<T>
{
    internal readonly ref T _reference;
    private readonly int _length;
    ...
}
```

The rollout of `ref` fields throughout [dotnet/runtime](https://github.com/dotnet/runtime) was done in [dotnet/runtime#71498](https://github.com/dotnet/runtime/pull/71498), following the C# language gaining this support primarily in [dotnet/roslyn#62155](https://github.com/dotnet/roslyn/pull/62155), which itself was the culmination of many PRs first into a feature branch. `ref` fields alone doesn't itself automatically improve performance, but it does simplify code significantly, and it allows for both new custom code that uses `ref` fields as well as new APIs that take advantage of them, both of which can help with performance (and specifically performance without sacrificing potential safety). One such example of a new API is new constructors on `ReadOnlySpan<T>` and `Span<T>`:

```csharp
public Span(ref T reference);
public ReadOnlySpan(in T reference);
```

added in [dotnet/runtime#67447](https://github.com/dotnet/runtime/pull/67447) (and then made public and used more broadly in [dotnet/runtime#71589](https://github.com/dotnet/runtime/pull/71589)). This may beg the question, why does `ref` field support enable two new constructors that take `ref`s, considering spans already were able to store a `ref`? After all, the `MemoryMarshal.CreateSpan(ref T reference, int length)` and corresponding `CreateReadOnlySpan` methods have existed for as long as spans have, and these new constructors are equivalent to calling those methods with a length of 1. The answer is: safety.

Imagine if you could willy-nilly call this constructor. You'd be able to write code like this:

```csharp
public Span<int> RuhRoh()
{
    int i = 42;
    return new Span<int>(ref i);
}
```

At this point the caller of this method is handed a span that refers to garbage; that's bad in code that's intended to be safe. You can already accomplish the same thing by using pointers:

```csharp
public Span<int> RuhRoh()
{
    unsafe
    {
        int i = 42;
        return new Span<int>(&i, 1);
    }
}
```

but at that point you've taken on the risk of using unsafe code and pointers and any resulting problems are on you. With C# 11, if you now try to write the above code using the `ref`-based constructor, you'll be greeted with an error like this:

```
error CS8347: Cannot use a result of 'Span<int>.Span(ref int)' in this context because it
may expose variables referenced by parameter 'reference' outside of their declaration scope
```

In other words, the compiler now understands that `Span<int>` as a `ref struct` could be storing the passed in `ref`, and if it does store it (which `Span<T>` does), this is akin to passing a `ref` to a local out of the method, which is bad. Hence how this relates to `ref` fields: because `ref` fields are now a thing, the compiler's rules for safe-handling of `ref`s have been updated, which in turn enables us to expose the aforementioned constructors on `{ReadOnly}Span<T>`.

As is often the case, addressing one issue kicks the can down the road a bit and exposes another. The compiler now believes that a `ref` passed to a method on a `ref struct` could enable that `ref struct` instance to store the `ref` (note that this was already the case with `ref struct`s passed to methods on `ref struct`s), but what if we don't want that? What if we want to be able to say "this `ref` is not storable and should not escape the calling scope"? From a caller's perspective, we want the compiler to allow passing in such `ref`s without it complaining about potential extension of lifetime, and from a callee's perspective, we want the compiler to prevent the method from doing what it's not supposed to do. Enter `scoped`. The new C# keyword does exactly what we just wished for: put it on a `ref` or `ref struct` parameter, and the compiler both will guarantee (short of using unsafe code) that the method can't stash away the argument and will then enable the caller to write code that relies on that guarantee. For example, consider this program:

```
var writer = new SpanWriter(stackalloc char[128]);
Append(ref writer, 123);
writer.Write(".");
Append(ref writer, 45);
Console.WriteLine(writer.AsSpan().ToString());

static void Append(ref SpanWriter builder, byte value)
{
    Span<char> tmp = stackalloc char[3];
    value.TryFormat(tmp, out int charsWritten);
    builder.Write(tmp.Slice(0, charsWritten));
}

ref struct SpanWriter
{
    private readonly Span<char> _chars;
    private int _length;

    public SpanWriter(Span<char> destination) => _chars = destination;

    public Span<char> AsSpan() => _chars.Slice(0, _length);

    public void Write(ReadOnlySpan<char> value)
    {
        if (_length > _chars.Length - value.Length)
        {
            throw new InvalidOperationException("Not enough remaining space");
        }

        value.CopyTo(_chars.Slice(_length));
        _length += value.Length;
    }
}
```

We have a `ref struct SpanWriter` that takes a `Span<char>` to its constructor and allows for writing to it by copying in additional content and then updating the stored length. The `Write` method accepts a `ReadOnlySpan<char>`. And we then have a helper `Append` method which is formatting a `byte` into some `stackalloc`'d temporary space and passing the resulting formatted `char`s in to `Write`. Straightforward. Except, this doesn't compile:

```
error CS8350: This combination of arguments to 'SpanWriter.Write(ReadOnlySpan<char>)' is
disallowed because it may expose variables referenced by parameter 'value' outside of their
declaration scope
```

What do we do? The `Write` method doesn't actually store the `value` parameter and won't ever need to, so we can change the signature of the method to annotate it as `scoped`:

```
public void Write(scoped ReadOnlySpan<char> value)
```

If `Write` were then to try to store `value`, the compiler would balk:

```
error CS8352: Cannot use variable 'ReadOnlySpan<char>' in this context because it may
expose referenced variables outside of their declaration scope
```

But as it's not trying to do so, everything now compiles successfully. You can see examples of how this is utilized in the aforementioned dotnet/runtime#71589.

There's also the other direction: there are some things that are implicitly `scoped`, like the `this` reference on a struct. Consider this code:

```
public struct SingleItemList
{
    private int _value;

    public ref int this[int i]
    {
        get
        {
            if (i != 0) throw new IndexOutOfRangeException();

            return ref _value;
        }
    }
}
```

This produces a compiler error:

```
error CS8170: Struct members cannot return 'this' or other instance members by reference
```

Effectively, that's because `this` is implicitly `scoped` (even though that keyword wasn't previously available). What if we want to enable such an item to be returned? Enter `[UnscopedRef]`. This is rare enough in need that it doesn't get its own C# language keyword, but the C# compiler does recognize the new `[UnscopedRef]` attribute. It can be put onto relevant parameters but also onto methods and properties, in which case it applies to the `this` reference for that member. As such, we can modify our previous code example to be:

```
[UnscopedRef]
public ref int this[int i]
```

and now the code will compile successfully. Of course, this also places demands on callers of this method. For a call site, the compiler sees the `[UnscopedRef]` on the member being invoked, and then knows that the returned `ref` might reference something from that struct, and thus assigns to the returned `ref` the same lifetime as that struct. So, if that struct were a local living on the stack, the `ref` would also be limited to that same method.

Another impactful span-related change comes in [dotnet/runtime#70095](https://github.com/dotnet/runtime/pull/70095) from [@teo-tsirpanis](https://github.com/teo-tsirpanis). `System.HashCode`'s goal is to provide a fast, easy-to-use implementation for producing high-quality hash codes. In its current incarnation, it incorporates a random process-wide seed and is an implementation of the xxHash32 non-cryptographic hash algorithm. In a previous release, `HashCode` saw the addition of an `AddBytes` methods, which accepts a `ReadOnlySpan<byte>` and is useful for incorporating sequences of data that should be part of a type's hash code, e.g. `BigInteger.GetHashCode` includes all the data that makes up the `BigInteger`. The xxHash32 algorithm works by accumulating 4 32-bit unsigned integers and then combining them together into the hash code; thus if you call `HashCode.Add(int)`, the first three times you call it you're just storing the values separately into the instance, and then the fourth time you call it all of those values are combined into the hash code (and there's a separate process that incorporates any remaining values if the number of 32-bit values added wasn't an exact multiple of 4). Thus, previously `AddBytes` was simply implemented to repeatedly read the next 4 bytes from the input span and call `Add(int)` with those bytes as an integer. But those `Add` calls have overhead. Instead, this PR skips the `Add` calls and directly handles the accumulation and combining of the 16 bytes. Interestingly, it still has to deal with the possibility that previous calls to `Add` may have left some state queued, which means (with the current implementation at least), if there are multiple pieces of state to include in the hash code, say a `ReadOnlySpan<byte>` and an additional `int`, it's more efficient to add the span first and then the `int` rather than the other way around. So for example when [dotnet/runtime#71274](https://github.com/dotnet/runtime/pull/71274) from [@huoyaoyuan](https://github.com/huoyaoyuan) changed `BigInteger.GetHashCode` to use `HashCode.AddBytes`, it coded the method to first call `AddBytes` with the `BigInteger`'s `_bits` and *then* call `Add` with the `_sign`.

```
private byte[] _data = Enumerable.Range(0, 256).Select(i => (byte)i).ToArray();

[Benchmark]
public int AddBytes()
{
    HashCode hc = default;
    hc.AddBytes(_data);
    return hc.ToHashCode();
}
```

Method	Runtime	Mean	Ratio
AddBytes	.NET 6.0	159.11 ns	1.00
AddBytes	.NET 7.0	42.11 ns	0.26

Another span-related change, [dotnet/runtime#72727](https://github.com/dotnet/runtime/pull/72727) refactored a bunch of code paths to eliminate some cached arrays. Why avoid cached arrays? After all, isn't it desirable to cache an array once and reuse it over and over again? It is, if that's the best option, but sometimes there are better options. For example, one of the changes took code like:

```
private static readonly char[] s_pathDelims = { ':', '\\', '/', '?', '#' };
...
int index = value.IndexOfAny(s_pathDelims);
```

and replaced it with code like:

```
int index = value.AsSpan().IndexOfAny(@":\/?#");
```

This has a variety of benefits. There's the usability benefit of keeping the tokens being searched close to the use site, and the usability benefit of the list being immutable such that some code somewhere won't accidentally replace a value in the array. But there are also performance benefits. We don't need an extra field to store the array. We don't need to allocate the array as part of this type's static constructor. And loading/using the string is slightly faster.

```
private static readonly char[] s_pathDelims = { ':', '\\', '/', '?', '#' };
private static readonly string s_value = "abcdefghijklmnopqrstuvwxyz";

[Benchmark]
public int WithArray() => s_value.IndexOfAny(s_pathDelims);

[Benchmark]
public int WithString() => s_value.AsSpan().IndexOfAny(@":\/?#");
```

Method	Mean	Ratio
WithArray	8.601 ns	1.00
WithString	6.949 ns	0.81

Another example from that PR took code along the lines of:

```
private static readonly char[] s_whitespaces = new char[] { ' ', '\t', '\n', '\r' };
...
switch (attr.Value.Trim(s_whitespaces))
{
    case "preserve": return Preserve;
    case "default": return Default;
}
```

and replaced it with code like:

```
switch (attr.Value.AsSpan().Trim(" \t\n\r"))
{
    case "preserve": return Preserve;
    case "default": return Default;
}
```

In this case, not only have we avoided the `char[]`, but if the text did require any trimming of whitespaces, the new version (which trims a span instead of the original string) will save an allocation for the trimmed string. This is taking advantage of the new C# 11 feature that supports switching on `ReadOnlySpan<char>`s just as you can switch on `string`s, added in [dotnet/roslyn#44388](https://github.com/dotnet/roslyn/pull/44388) from [@YairHalberstadt](https://github.com/YairHalberstadt). dotnet/runtime#68831 also took advantage of this in several additional places.

Of course, in some cases the arrays are entirely unnecessary. In that same PR, there were several cases like this:

```
private static readonly char[] WhiteSpaceChecks = new char[] { ' ', '\u00A0' };
...
int wsIndex = target.IndexOfAny(WhiteSpaceChecks, targetPosition);
if (wsIndex < 0)
{
    return false;
}
```

By switching to use spans, again, we can instead write it like this:

```
int wsIndex = target.AsSpan(targetPosition).IndexOfAny(' ', '\u00A0');
if (wsIndex < 0)
{
    return false;
}
wsIndex += targetPosition;
```

`MemoryExtensions.IndexOfAny` has a dedicated overload for two and three arguments, at which point we don't need the array at all (these overloads also happen to be faster; when passing an array of two chars, the implementation would extract the two chars from the array and pass them off to the same two-argument implementation). Multiple other PRs similarly removed array allocations. dotnet/runtime#60409 removed a single-char array that was cached to be able to pass it to `string.Split` and replaced it with usage of the `Split` overload that directly accepts a single `char`.

Finally, dotnet/runtime#59670 from [@NewellClark](https://github.com/NewellClark) got rid of even more arrays. We saw earlier how the C# compiler special-cases `byte[]`s constructed with a constant length and constant elements and that's immediately cast to a `ReadOnlySpan<byte>`. Thus, it can be beneficial any time there's such a `byte[]` being cached to instead expose it as a `ReadOnlySpan<byte>`. As I discussed in the .NET 6 post, this avoids even the one-time array allocation you'd get for a cached array, results in much more efficient access, and supplies to the JIT compiler more information that enables it to more heavily optimize… goodness all around. This PR removed even more arrays in this manner, as did dotnet/runtime#60411, dotnet/runtime#72743, dotnet/runtime#73115 from [@vcsjones](https://github.com/vcsjones), and dotnet/runtime#70665.

CHAPTER 11

Regex

Back in May, I shared a fairly detailed post about the improvements coming to Regular Expressions in .NET 7. As a recap, prior to .NET 5, `Regex`'s implementation had largely been untouched for quite some time. In .NET 5, we brought it back up to be on par with or better than multiple other industry implementations from a performance perspective. .NET 7 takes some significant leaps forward from that. If you haven't read the post yet, please go ahead and do so now; I'll wait…

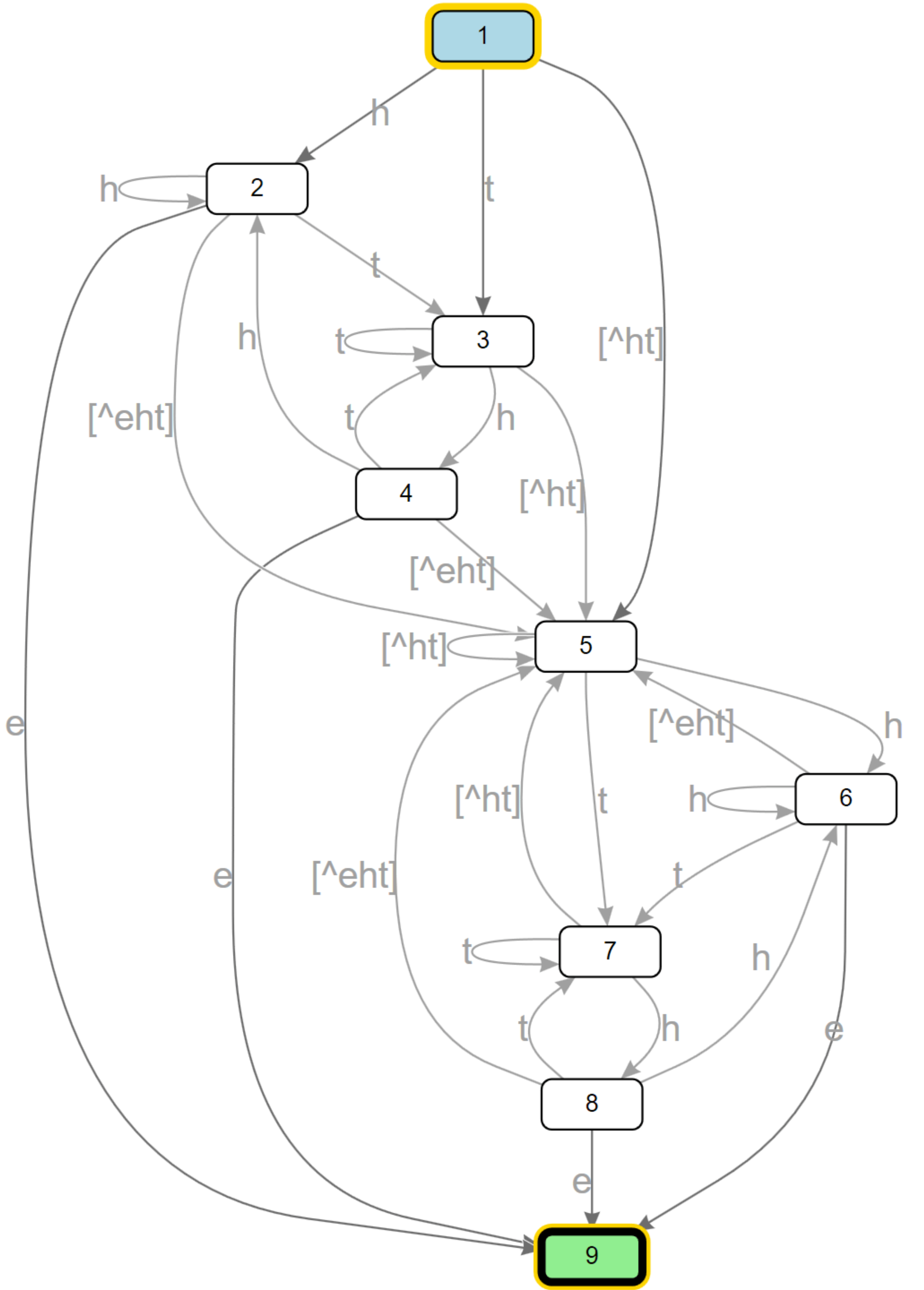
Welcome back. With that context, I'll avoid duplicating content here, and instead focus on how exactly these improvements came about and the PRs that did so.

RegexOptions.NonBacktracking

Let's start with one of the larger new features in `Regex`, the new `RegexOptions.NonBacktracking` implementation. As discussed in the previous post, `RegexOptions.NonBacktracking` switches the processing of `Regex` over to using a new engine based in finite automata. It has two primary modes of execution, one that relies on DFAs (deterministic finite automata) and one that relies on NFAs (non-deterministic finite automata). Both implementations provide a very valuable guarantee: processing time is linear in the length of the input. Whereas a backtracking engine (which is what `Regex` uses if `NonBacktracking` isn't specified) can hit a situation known as "catastrophic backtracking," where problematic expressions combined with problematic input can result in exponential processing in the length of the input, `NonBacktracking` guarantees it'll only ever do an ammortized-constant amount of work per character in the input. In the case of a DFA, that constant is very small. With an NFA, that constant can be much larger, based on the complexity of the pattern, but for any given pattern the work is still linear in the length of the input.

A significant number of years of development went into the `NonBacktracking` implementation, which was initially added into dotnet/runtime in dotnet/runtime#60607. However, the original research and implementation for it actually came from Microsoft Research (MSR), and was available as an experimental package in the form of the Symbolic Regex Matcher (SRM) library published by MSR. You can still see vestiges of this in the current code now in .NET 7, but it's evolved significantly, in tight collaboration between developers on the .NET team and the researchers at MSR (prior to being integrated in dotnet/runtime, it was incubated for over a year in dotnet/runtimelab, where the original SRM code was brought in via dotnet/runtimelab#588 from [@veanes](https://github.com/veanes)).

This implementation is based on the notion of regular expression derivatives, a concept that's been around for decades (the term was originally coined in a paper by Janusz Brzozowski in the 1960s) and which has been significantly advanced for this implementation. Regex derivatives form the basis for how the automata (think "graph") used to process input are constructed. The idea at its core is fairly simple: take a regex and process a single character… what is the new regex you get to describe what remains after processing that one character? That's the derivative. For example, given the regex `\w{3}` to match three word characters, if you apply this to the next input character 'a', well, that will strip off the first `\w`, leaving us with the derivative `\w{2}`. Simple, right? How about something more complicated, like the expression `.*(the|he)`. What happens if the next character is a `t`? Well, it's possible that `t` could be consumed by the `.*` at the beginning of the pattern, in which case the remaining regex would be exactly the same as the starting one (`.*(the|he)`), since after matching `t` we could still match exactly the same input as without the `t`. But, the `t` could have also been part of matching `the`, and applied to `the`, we'd strip off the `t` and be left with `he`, so now our derivative is `.*(the|he)|he`. Then what about the `he` in the original alternation? `t` doesn't match `h`, so the derivative would be nothing, which we'll express here as an empty character class, giving us `.*(the|he)|he|[]`. Of course, as part of an alternation, that "nothing" at the end is a nop, and so we can simplify the whole derivative to just `.*(the|he)|he`… done. That was all when applying the original pattern against a next `t`. What if it was against an `h` instead? Following the same logic as for the `t`, this time we end up with `.*(the|he)|e`. And so on. What if we instead start with the `h` derivative and the next character is an `e`? Then we're taking the pattern `.*(the|he)|e` and applying it to `e`. Against the left side of the alternation, it can be consumed by the `.*` (but doesn't match either `t` or `h`), and so we just end up with that same subexpression. But against the right side of the alternation, `e` matches `e`, leaving us with the empty string `()`: `.*(the|he)|()`. At the point where a pattern is "nullable" (it can match the empty string), that can be considered a match. We can visualize this whole thing as a graph, with transitions for every input character to the derivative that comes from applying it.


1
h
t
2
h
t
3
t
h
[^ht]
[^eht]
t
h
4
[^ht]
[^eht]
5
[^ht]
[^eht]
h
e
[^ht]
t
h
6
e
[^eht]
t
7
t
h
t
h
e
8
e
9

Looks an awful lot like a DFA, doesn't it? It should. And that's exactly how `NonBacktracking` constructs the DFAs it uses to process input. For every regex construct (concatenations, alternations, loops, etc.) the engine knows how to derive the next regex based on the character being evaluated. This application is done lazily, so we have an initial starting state (the original pattern), and then when we evaluate the next character in the input, it looks to see whether there's already a derivative available for that transition: if there is, it follows it, and if there isn't, it dynamically/lazily derives the next node in the graph. At its core, that's how it works.

Of course, the devil is in the details and there's a ton of complication and engineering smarts that go into making the engine efficient. One such example is a tradeoff between memory consumption and throughput. Given the ability to have any `char` as input, you could have effectively ~65K transitions out of every node (e.g. every node could need a ~65K element table); that would significantly increase memory consumption. However, if you actually had that many transitions, it's very likely a significant majority of them would point to the same target node. Thus, `NonBacktracking` maintains its own groupings of characters into what it calls "minterms." If two characters will have exactly the same transition, they're part of the same minterm. The transitions are then constructed in terms of minterms, with at most one transition per minterm out of a given node. When the next input character is read, it maps that to a minterm ID, and then finds the appropriate transition for that ID; one additional level of indirection in order to save a potentially huge amount of memory. That mapping is handled via an array bitmap for ASCII and an efficient data structure known as a [Binary Decision Diagram (BDD)]() for everything above 0x7F.

As noted, the non-backtracking engine is linear in the length of the input. But that doesn't mean it always looks at each input character exactly once. If you call `Regex.IsMatch`, it does; after all, `IsMatch` only needs to determine whether there is a match and doesn't need to compute any additional information, such as where the match actual starts or ends, any information on captures, etc. Thus, the engine can simply employ its automata to walk along the input, transitioning from node to node in the graph until it comes to a final state or runs out of input. Other operations, however, do require it to gather more information. `Regex.Match` needs to compute everything, and that can actually entail multiple walks over the input. In the initial implementation, the equivalent of `Match` would always take three passes: match forwards to find the end of *a* match, then match a reversed-copy of the pattern in reverse from that ending location in order to find where the match actually starts, and then once more walk forwards from that known starting position to find the actual ending position. However, with [dotnet/runtime#68199]() from [@olsaarik](https://github.com/olsaarik), unless captures are required, it can now be done in only two passes: once forward to find the guaranteed ending location of the match, and then once in reverse to find its starting location. And [dotnet/runtime#65129]() from [@olsaarik](https://github.com/olsaarik) added captures support, which the original implementation also didn't have. This captures support adds back a third pass, such that once the bounds of the match are known, the engine runs the forward pass one more time, but this time with an NFA-based "simulation" that is able to record "capture effects" on transitions. All of this enables the non-backtracking implementation to have the exact same semantics as the backtracking engines, always producing the same matches in the same order with the same capture information. The only difference in this regard is, whereas with the backtracking engines capture groups inside of loops will store all values captured in every iteration of the loop, only the last iteration is stored with the non-backtracking implementation. On top of that, there are a few constructs the non-backtracking implementation simply doesn't support, such that attempting to use any of those will fail when trying to construct the `Regex`, e.g. backreferences and lookarounds.

Even after its progress as a standalone library from MSR, more than 100 PRs went into making `RegexOptions.NonBacktracking` what it is now in .NET 7, including optimizations like [dotnet/runtime#70217](https://github.com/dotnet/runtime/pull/70217) from [@olsaarik](https://github.com/olsaarik) that tries to streamline the tight inner matching loop at the heart of the DFA (e.g. read the next input character, find the appropriate transition to take, move to the next node, and check information about the node like whether it's a final state) and optimizations like [dotnet/runtime#65637](https://github.com/dotnet/runtime/pull/65637) from [@veanes](https://github.com/veanes) that optimized the NFA mode to avoid superfluous allocations, caching and reusing list and set objects to make the handling of the lists of states ammortized allocation-free.

There's one more set of PRs of performance interest for `NonBacktracking`. The `Regex` implementation for taking patterns and turning them into something processable, regardless of which of the multiple engines is being used, is essentially a compiler, and as with many compilers, it naturally lends itself to recursive algorithms. In the case of `Regex`, those algorithms involve walking around trees of regular expression constructs. Recursion ends up being a very handy way of expressing these algorithms, but recursion also suffers from the possibility of stack overflow; essentially it's using stack space as scratch space, and if it ends up using too much, things go badly. One common approach to dealing with this is turning the recursive algorithm into an iterative one, which typically involves using an explicit stack of state rather than the implicit one. The nice thing about this is the amount of state you can store is limited only by how much memory you have, as opposed to being limited by your thread's stack space. The downsides, however, are that it's typically much less natural to write the algorithms in this manner, and it typically requires allocating heap space for the stack, which then leads to additional complications if you want to avoid that allocation, such as various kinds of pooling. [dotnet/runtime#60385](https://github.com/dotnet/runtime/pull/60385) introduces a different approach for `Regex`, which is then used by [dotnet/runtime#60786](https://github.com/dotnet/runtime/pull/60786) from [@olsaarik](https://github.com/olsaarik) specifically in the `NonBacktracking` implementation. It still uses recursion, and thus benefits from the expressiveness of the recursive algorithm as well as being able to use stack space and thus avoid additional allocation in the most common cases, but then to avoid stack overflows, it issues explicit checks to ensure we're not too deep on the stack (.NET has long provided the helpers `RuntimeHelpers.EnsureSufficientExecutionStack` and `RuntimeHelpers.TryEnsureSufficientExecutionStack` for this purpose). If it detects it's too deep on the stack, it forks off continued execution into another thread. Hitting this condition is expensive, but it's very rarely if ever actually hit in practice (e.g. the only time it's hit in our vast functional tests are in the tests explicitly written to stress it), it keeps the code simple, and it keeps the typical cases fast. A similar approach is used in other areas of [dotnet/runtime](https://github.com/dotnet/runtime), such as in System.Linq.Expressions.

As was mentioned in my previous blog post about regular expressions, both the backtracking implementations and the non-backtracking implementation have their place. The main benefit of the non-backtracking implementation is predictability: because of the linear processing guarantee, once you've constructed the regex, you don't need to worry about malicious inputs causing worst-case behavior in the processing of your potentially susceptible expressions. This doesn't mean `RegexOptions.NonBacktracking` is always the fastest; in fact, it's frequently not. In exchange for reduced best-case performance, it provides the best worst-case performance, and for some kinds of applications, that's a really worthwhile and valuable tradeoff.

New APIs

`Regex` gets several new methods in .NET 7, all of which enable improved performance. The simplicity of the new APIs likely also misrepresents how much work was necessary to enable them, in particular because the new APIs all support `ReadOnlySpan<char>` inputs into the regex engines.

dotnet/runtime#65473 brings `Regex` into the span-based era of .NET, overcoming a significant limitation in `Regex` since spans were introduced back in .NET Core 2.1. `Regex` has historically been based on processing `System.String` inputs, and that fact pervades the `Regex` design and implementation, including the APIs exposed for the extensibility model `Regex.CompileToAssembly` relied on in .NET Framework (`CompileToAssembly` is now obsoleted and has never been functional in .NET Core). One subtly that relies on the nature of `string` as the input is how match information is returned to callers. `Regex.Match` returns a `Match` object that represents the first match in the input, and that `Match` object exposes a `NextMatch` method that enables moving to the next match. That means the `Match` object needs to store a reference to the input, so that it can be fed back into the matching engine as part of such a `NextMatch` call. If that input is a `string`, great, no problem. But if that input is a `ReadOnlySpan<char>`, that span as a `ref struct` can't be stored on the `class Match` object, since `ref structs` can only live on the stack and not the heap. That alone would make it a challenge to support spans, but the problem is even more deeply rooted. All of the regex engines rely on a `RegexRunner`, a base class that stores on it all of the state necessary to feed into the `FindFirstChar` and `Go` methods that compose the actual matching logic for the regular expressions (these methods contain all of the core code for performing the match, with `FindFirstChar` being an optimization to skip past input positions that couldn't possibly start a match and then `Go` performing the actual matching logic). If you look at the internal `RegexInterpreter` type, which is the engine you get when you construct a `new Regex(...)` without the `RegexOptions.Compiled` or `RegexOptions.NonBacktracking` flags, it derives from `RegexRunner`. Similarly, when you use `RegexOptions.Compiled`, it hands off the dynamic methods it reflection emits to a type derived from `RegexRunner`, `RegexOptions.NonBacktracking` has a `SymbolicRegexRunnerFactory` that produces types derived from `RegexRunner`, and so on. Most relevant here, `RegexRunner` is public, because the types generated by the `Regex.CompileToAssembly` type (and now the regex source generator) include ones derived from this `RegexRunner`. Those `FindFirstChar` and `Go` methods are thus `abstract` and `protected`, and parameterless, because they pick up all the state they need from protected members on the base class. That includes the `string` input to process. So what about spans? We could of course have just called `ToString()` on an input `ReadOnlySpan<char>`. That would have been functionally correct, but would have completely defeated the purpose of accepting spans, and worse, would have been so unexpected as to likely cause consuming apps to be worse performing than they would have without the APIs. Instead, we needed a new approach and new APIs.

First, we made `FindFirstChar` and `Go` virtual instead of abstract. The design that splits these methods is largely antiquated, and in particular the forced separation between a stage of processing where you find the next possible location of a match and then a stage where you actually perform the match at that location doesn't align well with all engines, like the one used by `NonBacktracking` (which initially implemented `FindFirstChar` as a nop and had all its logic in `Go`). Then we added a new virtual `Scan` method which, importantly, takes a `ReadOnlySpan<char>` as a parameter; the span can't be exposed from the base `RegexRunner` and must be passed in. We then implemented `FindFirstChar` and `Go` in terms of `Scan`, and made them "just work." Then, all of the engines are implemented in terms of that span; they no longer need to access the protected `RegexRunner.runtext`, `RegexRunner.runtextbeg`, and `RegexRunner.runtextend` members that surface the input; they're just handed the span, already sliced to the input region, and process that. One of the neat things about this from a performance perspective is it enables the JIT to do a better job at shaving off various overheads, in particular around bounds checking. When the logic is implemented in terms of `string`, in addition to the input string itself the engine is also handed the beginning and end of the region of the input to process (since the developer could have called a method like `Regex.Match(string input, int beginning, int length)` in order to only process a substring). Obviously the engine matching logic is way more complicated than this, but simplifying, imagine the entirety of the engine was just a loop over the input. With the input, beginning, and length, that would look like:

```
[Benchmark]
[Arguments("abc", 0, 3)]
public void Scan(string input, int beginning, int length)
{
    for (int i = beginning; i < length; i++)
    {
        Check(input[i]);
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void Check(char c) { }
```

That will result in the JIT generating assembly code along the lines of this:

```
; Program.Scan(System.String, Int32, Int32)
       sub       rsp,28
       cmp       r8d,r9d
       jge       short M00_L01
       mov       eax,[rdx+8]
M00_L00:
       cmp       r8d,eax
       jae       short M00_L02
       inc       r8d
       cmp       r8d,r9d
       jl        short M00_L00
M00_L01:
       add       rsp,28
       ret
M00_L02:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 36
```

In contrast, if we're dealing with a span, which already factors in the bounds, then we can write a more canonical loop like this:

```
[Benchmark]
[Arguments("abc")]
public void Scan(ReadOnlySpan<char> input)
{
    for (int i = 0; i < input.Length; i++)
    {
        Check(input[i]);
    }
}
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void Check(char c) { }
```

And when it comes to compilers, something in a canonical form is really good, because the more common the shape of the code, the more likely it is to be heavily optimized:

```
; Program.Scan(System.ReadOnlySpan`1<Char>)
       mov       rax,[rdx]
       mov       edx,[rdx+8]
       xor       ecx,ecx
       test      edx,edx
       jle       short M00_L01
M00_L00:
       mov       r8d,ecx
       movsx     r8,word ptr [rax+r8*2]
       inc       ecx
       cmp       ecx,edx
       jl        short M00_L00
M00_L01:
       ret
; Total bytes of code 27
```

So even without all the other benefits that come from operating in terms of span, we immediately get low-level code generation benefits from performing all the logic in terms of spans. While the above example was made up (obviously the matching logic does more than a simple for loop), here's a real example. When a regex contains a `\b`, as part of evaluating the input against that `\b` the backtracking engines call a `RegexRunner.IsBoundary` helper method which checks whether the character at the current position is a word character and whether the character before it is a word character (factoring in the bounds of the input as well). Here's what the `IsBoundary` method based on `string` looked like (the `runtext` it's using is the name of the `string` field on `RegexRunner` that stores the input):

```
[Benchmark]
[Arguments(0, 0, 26)]
public bool IsBoundary(int index, int startpos, int endpos)
{
    return (index > startpos && IsBoundaryWordChar(runtext[index - 1])) !=
           (index < endpos   && IsBoundaryWordChar(runtext[index]));
}

[MethodImpl(MethodImplOptions.NoInlining)]
private bool IsBoundaryWordChar(char c) => false;
```

and here's what the span version looks like:

```
[Benchmark]
[Arguments("abcdefghijklmnopqrstuvwxyz", 0)]
public bool IsBoundary(ReadOnlySpan<char> inputSpan, int index)
{
    int indexM1 = index - 1;
    return ((uint)indexM1 < (uint)inputSpan.Length &&
IsBoundaryWordChar(inputSpan[indexM1])) !=
            ((uint)index < (uint)inputSpan.Length && IsBoundaryWordChar(inputSpan[index]));
}

[MethodImpl(MethodImplOptions.NoInlining)]
private bool IsBoundaryWordChar(char c) => false;
```

And here's the resulting assembly:

```
; Program.IsBoundary(Int32, Int32, Int32)
       push      rdi
       push      rsi
       push      rbp
       push      rbx
       sub       rsp,28
       mov       rdi,rcx
       mov       esi,edx
       mov       ebx,r9d
       cmp       esi,r8d
       jle       short M00_L00
       mov       rcx,rdi
       mov       rcx,[rcx+8]
       lea       edx,[rsi-1]
       cmp       edx,[rcx+8]
       jae       short M00_L04
       mov       edx,edx
       movzx     edx,word ptr [rcx+rdx*2+0C]
       mov       rcx,rdi
       call      qword ptr [Program.IsBoundaryWordChar(Char)]
       jmp       short M00_L01
M00_L00:
       xor       eax,eax
M00_L01:
       mov       ebp,eax
       cmp       esi,ebx
       jge       short M00_L02
       mov       rcx,rdi
       mov       rcx,[rcx+8]
       cmp       esi,[rcx+8]
       jae       short M00_L04
       mov       edx,esi
       movzx     edx,word ptr [rcx+rdx*2+0C]
       mov       rcx,rdi
       call      qword ptr [Program.IsBoundaryWordChar(Char)]
       jmp       short M00_L03
M00_L02:
       xor       eax,eax
M00_L03:
       cmp       ebp,eax
       setne     al
       movzx     eax,al
       add       rsp,28
       pop       rbx
       pop       rbp
       pop       rsi
       pop       rdi
       ret
M00_L04:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 117

; Program.IsBoundary(System.ReadOnlySpan`1<Char>, Int32)
       push      r14
       push      rdi
       push      rsi
       push      rbp
```

```
        push      rbx
        sub       rsp,20
        mov       rdi,rcx
        mov       esi,r8d
        mov       rbx,[rdx]
        mov       ebp,[rdx+8]
        lea       edx,[rsi-1]
        cmp       edx,ebp
        jae       short M00_L00
        mov       edx,edx
        movzx     edx,word ptr [rbx+rdx*2]
        mov       rcx,rdi
        call      qword ptr [Program.IsBoundaryWordChar(Char)]
        jmp       short M00_L01
M00_L00:
        xor       eax,eax
M00_L01:
        mov       r14d,eax
        cmp       esi,ebp
        jae       short M00_L02
        mov       edx,esi
        movzx     edx,word ptr [rbx+rdx*2]
        mov       rcx,rdi
        call      qword ptr [Program.IsBoundaryWordChar(Char)]
        jmp       short M00_L03
M00_L02:
        xor       eax,eax
M00_L03:
        cmp       r14d,eax
        setne     al
        movzx     eax,al
        add       rsp,20
        pop       rbx
        pop       rbp
        pop       rsi
        pop       rdi
        pop       r14
        ret
; Total bytes of code 94
```

The most interesting thing to notice here is the:

```
call      CORINFO_HELP_RNGCHKFAIL
int       3
```

at the end of the first version that doesn't exist at the end of the second. As we saw earlier, this is what the generated assembly looks like when the JIT is emitting the code to throw an index out of range exception for an array, string, or span. It's at the end because it's considered to be "cold," rarely executed. It exists in the first because the JIT can't prove based on local analysis of that function that the `runtext[index-1]` and `runtext[index]` accesses will be in range of the string (it can't know or trust any implied relationship between `startpos`, `endpos`, and the bounds of `runtext`). But in the second, the JIT can know and trust that the `ReadOnlySpan<char>`'s lower bound is 0 and upper bound (exclusive) is the span's `Length`, and with how the method is constructed, it can then prove that the span accesses are always in bound. As such, it doesn't need to emit any bounds checks in the method, and the method then lacks the tell-tale signature of the index out of range throw. You can see more examples of taking advantage of spans now being at the heart of the all of the engines in dotnet/runtime#66129, dotnet/runtime#66178, and dotnet/runtime#72728, all of which clean up unnecessary checks against the bounds that are then always `0` and `span.Length`.

Ok, so the engines are now able to be handed span inputs and process them, great, what can we do with that? Well, `Regex.IsMatch` is easy: it's not encumbered by needing to perform multiple matches, and thus doesn't need to worry about how to store that input `ReadOnlySpan<char>` for the next match. Similarly, the new `Regex.Count`, which provides an optimized implementation for counting how many matches there are in the input, can bypass using `Match` or `MatchCollection`, and thus can easily operate over spans as well; dotnet/runtime#64289 added `string`-based overloads, and dotnet/runtime#66026 added span-based overloads. We can optimize `Count` further by passing additional information into the engines to let them know how much information they actually need to compute. For example, I noted previously that `NonBacktracking` is fairly pay-for-play in how much work it needs to do relative to what information it needs to gather. It's cheapest to just determine whether there is a match, as it can do that in a single forward pass through the input. If it also needs to compute the actual starting and ending bounds, that requires another reverse pass through some of the input. And if it then also needs to compute capture information, that requires yet another forward pass based on an NFA (even if the other two were DFA-based). `Count` needs the bounds information, as it needs to know where to start looking for the next match, but it doesn't need the capture information, since none of that capture information is handed back to the caller. dotnet/runtime#68242 updates the engines to receive this additional information, such that methods like `Count` can be made more efficient.

So, `IsMatch` and `Count` can work with spans. But we still don't have a method that lets you actually get back that match information. Enter the new `EnumerateMatches` method, added by dotnet/runtime#67794. `EnumerateMatches` is very similar to `Match`, except instead of handing back a `Match` class instance, it hands back a `ref struct` enumerator:

```
public ref struct ValueMatchEnumerator
{
    private readonly Regex _regex;
    private readonly ReadOnlySpan<char> _input;
    private ValueMatch _current;
    private int _startAt;
    private int _prevLen;
    ...
}
```

Being a `ref struct`, the enumerator is able to store a reference to the input span, and is thus able to iterate through matches, which are represented by the `ValueMatch` ref struct. Notably, today `ValueMatch` doesn't provide capture information, which also enables it to partake in the optimizations previously mentioned for `Count`. Even if you have an input `string`, `EnumerateMatches` is thus a way to have ammortized allocation-free enumeration of all matches in the input. In .NET 7, though, there isn't a way to have such allocation-free enumeration if you also need all the capture data. That's something we'll investigate designing in the future if/as needed.

TryFindNextPossibleStartingPosition

As noted earlier, the core of all of the engines is a `Scan(ReadOnlySpan<char>)` method that accepts the input text to match, combines that with positional information from the base instance, and exits when it either finds the location of the next match or exhausts the input without finding another. For the backtracking engines, the implementation of that method is logically as follows:

```
protected override void Scan(ReadOnlySpan<char> inputSpan)
{
    while (!TryMatchAtCurrentPosition(inputSpan) &&
           base.runtextpos != inputSpan.Length)
    {
        base.runtextpos++;
    }
}
```

We try to match the input at the current position, and if we're successful in doing so, that's it, we exit. If the current position doesn't match, however, then if there's any input remaining we "bump" the position and start the process over. In regex engine terminology, this is often referred to as a "bumpalong loop." However, if we actually ran the full matching process at every input character, that could be unnecessarily slow. For many patterns, there's something about the pattern that would enable us to be more thoughtful about where we perform full matches, quickly skipping past locations that couldn't possibly match, and only spending our time and resources on locations that have a real chance of matching. To elevate that concept to a first-class one, the backtracking engines' "bumpalong loop" is typically more like the following (I say "typically" because in some cases the compiled and source generated regexes are able to generate something even better).

```
protected override void Scan(ReadOnlySpan<char> inputSpan)
{
    while (TryFindNextPossibleStartingPosition(inputSpan) &&
           !TryMatchAtCurrentPosition(inputSpan) &&
           base.runtextpos != inputSpan.Length)
    {
        base.runtextpos++;
    }
}
```

As with `FindFirstChar` previously, that `TryFindNextPossibleStartingPosition` has the responsibility of searching as quickly as possible for the next place to match (or determining that nothing else could possibly match, in which case it would return `false` and the loop would exit). As `FindFirstChar`, and it was embued with multiple ways of doing its job. In .NET 7, `TryFindNextPossibleStartingPosition` learns many more and improved ways of helping the engine be fast.

In .NET 6, the interpreter engine had effectively two ways of implementing `TryFindNextPossibleStartingPosition`: a Boyer-Moore substring search if the pattern began with a string (potentially case-insensitive) of at least two characters, and a linear scan for a character class known to be the set of all possible chars that could begin a match. For the latter case, the interpreter had eight different implementations for matching, based on a combination of whether `RegexOptions.RightToLeft` was set or not, whether the character class required case-insensitive comparison or not, and whether the character class contained only a single character or more than one character. Some of these were more optimized than others, e.g. a left-to-right, case-sensitive, single-char search would use an `IndexOf(char)` to search for the next location, an optimization added in .NET 5. However, every time this operation was performed, the engine would need to recompute which case it would be. [dotnet/runtime#60822](https://github.com/dotnet/runtime/pull/60822) improved this, introducing an internal enum of the strategies used by `TryFindNextPossibleStartingPosition` to find the next opportunity, adding a `switch` to `TryFindNextPossibleStartingPosition` to quickly jump to the right strategy, and precomputing which strategy to use when the interpreter was constructed. This not only made the interpreter's implementation at match time faster, it made it effectively free (in terms of runtime overhead at match time) to add additional strategies.

[dotnet/runtime#60888](https://github.com/dotnet/runtime/pull/60888) then added the first additional strategy. The implementation was already capable of using `IndexOf(char)`, but as mentioned previously in this post, the implementation of `IndexOf(ReadOnlySpan<char>)` got way better in .NET 7 in many cases, to the point where it ends up being significantly better than Boyer-Moore in all but the most corner of corner cases. So this PR enables a new `IndexOf(ReadOnlySpan<char>)` strategy to be used to search for a prefix string in the case where the string is case-sensitive.

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;
private Regex _regex = new Regex(@"\belementary\b", RegexOptions.Compiled);

[Benchmark]
public int Count() => _regex.Matches(s_haystack).Count;
```

Method	Runtime	Mean	Ratio
Count	.NET 6.0	377.32 us	1.00
Count	.NET 7.0	55.44 us	0.15

[dotnet/runtime#61490](https://github.com/dotnet/runtime/pull/61490) then removed Boyer-Moore entirely. This wasn't done in the previously mentioned PR because of lack of a good way to handle case-insensitive matches. However, this PR also special-cased ASCII letters to teach the optimizer how to turn an ASCII case-insensitive match into a set of both casings of that letter (excluding the few known to be a problem, like `i` and `k`, which can both be impacted by the employed culture and which might map case-insensitively to more than two values). With enough of the common cases covered, rather than use Boyer-Moore to perform a case-insensitive search, the implementation just uses `IndexOfAny(char, char, ...)` to search for the starting set, and the vectorization employed by `IndexOfAny` ends up outpacing the old implementation handily in real-world cases. This PR goes further than that, such that it doesn't just discover the "starting set," but is able to find all of the character classes that could match a pattern a fixed-offset from the beginning; that then gives the analyzer the ability to choose the set that's expected to be least common and issue a search for it instead of whatever happens to be at the beginning. The PR goes even further, too, motivated in large part by the non-backtracking engine. The non-backtracking engine's prototype implementation also used `IndexOfAny(char, char, ...)` when it arrived at a starting state and was thus able to quickly skip through input text that wouldn't have a chance of pushing it to the next state. We wanted all of the engines to share as much logic as possible, in particular around this speed ahead, and so this PR unified the interpreter with the non-backtracking engine to have them share the exact same `TryFindNextPossibleStartingPosition` routine (which the non-backtracking engine just calls at an appropriate place in its graph traversal loop). Since the non-backtracking engine was already using `IndexOfAny` in this manner, initially not doing so popped as a significant regression on a variety of patterns we measure, and this caused us to invest in using it everywhere. This PR also introduced the first special-casing for case-insensitive comparisons into the compiled engine, e.g. if we found a set that was `[Ee]`, rather than emitting a check akin to `c == 'E' || c == 'e'`, we'd instead emit a check akin to `(c | 0x20) == 'e'` (those fun ASCII tricks discussed earlier coming into play again).

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;
private Regex _regex = new Regex(@"\belementary\b", RegexOptions.Compiled |
RegexOptions.IgnoreCase);

[Benchmark]
public int Count() => _regex.Matches(s_haystack).Count;
```

Method	Runtime	Mean	Ratio
Count	.NET 6.0	499.3 us	1.00
Count	.NET 7.0	177.7 us	0.35

The previous PR started turning `IgnoreCase` pattern text into sets, in particular for ASCII, e.g. `(?i)a` would become `[Aa]`. That PR hacked in the support for ASCII knowing that something more complete would be coming along, as it did in dotnet/runtime#67184. Rather than hardcoding the case-insensitive sets that just the ASCII characters map to, this PR essentially hardcodes the sets for every possible char. Once that's done, we no longer need to know about case-insensitivity at match time and can instead just double-down on efficiently matching sets, which we already need to be able to do well. Now, I said it encodes the sets for every possible char; that's not entirely true. If it were true, that would take up a large amount of memory, and in fact, most of that memory would be wasted because the vast majority of characters don't participate in case conversion… there are only ~2,000 characters that we need to handle. As such, the implementation employs a three-tier table scheme. The first table has 64 elements, dividing the full range of `chars` into 64 groupings; of those 64 groups, 54 of them have no characters that participate in case conversion, so if we hit one of those entries, we can immediately stop the search. For the remaining 10 that do have at least one character in their range participating, the character and the value from the first table are used to compute an index into the second table; there, too, the majority of entries say that nothing participates in case conversion. It's only if we get a legitimate hit in the second table does that give us an index into the third table, at which location we can find all of the characters considered case-equivalent with the first.

dotnet/runtime#63477 (and then later improved in dotnet/runtime#66572) proceeded to add another searching strategy, this one inspired by nim-regex's literal optimizations. There are a multitude of regexes we track from a performance perspective to ensure we're not regressing in common cases and to help guide investments. One is the set of patterns in mariomka/regex-benchmark languages regex benchmark. One of those is for URIs: `(@"[\w]+://[^/\s?#]+[^\s?#]+(?:\?[^\s#]*)?(?:#[^\s]*)?"`. This pattern defies the thus-far enabled strategies for finding a next good location, as it's guaranteed to begin with a "word character" (`\w`), which includes ~50,000 of the ~65,000 possible characters; we don't have a good way of vectorizing a search for such a character class. However, this pattern is interesting in that it begins with a loop, and not only that, it's an upper-unbounded loop which our analysis will determine is atomic, because the character guaranteed to immediately follow the loop is a `':'`, which is itself not a word character, and thus there's nothing the loop could match and give up as part of backtracking that would match `':'`. That all lends itself to a different approach to vectorization: rather than trying to search for the `\w` character class, we can instead search for the substring `"://"`, and then once we find it, we can match backwards through as many `[\w]`s as we can find; in this case, the only constraint is we need to match at least one. This PR added that strategy, for a literal after an atomic loop, to all of the engines.

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;
private Regex _regex = new Regex(@"[\w]+://[^/\s?#]+[^\s?#]+(?:\?[^\s#]*)?(?:#[^\s]*)?",
RegexOptions.Compiled);

[Benchmark]
public bool IsMatch() => _regex.IsMatch(s_haystack); // Uri's in Sherlock Holmes? "Most
unlikely."
```

Method	Runtime	Mean	Ratio
IsMatch	.NET 6.0	4,291.77 us	1.000
IsMatch	.NET 7.0	42.40 us	0.010

Of course, as has been talked about elsewhere, the best optimizations aren't ones that make something faster but rather ones that make something entirely unnecessary. That's what [dotnet/runtime#64177](https://github.com/dotnet/runtime/pull/64177) does, in particular in relation to anchors. The .NET regex implementation has long had optimizations for patterns with a starting anchor: if the pattern begins with `^`, for example (and `RegexOptions.Multiline` wasn't specified), the pattern is rooted to the beginning, meaning it can't possibly match at any position other than `0`; as such, with such an anchor, `TryFindNextPossibleStartingPosition` won't do any searching at all. The key here, though, is being able to detect whether the pattern begins with such an anchor. In some cases, like `^abc$`, that's trivial. In other cases, like `^abc|^def`, the existing analysis had trouble seeing through that alternation to find the guaranteed starting `^` anchor. This PR fixes that. It also adds a new strategy based on discovering that a pattern has an ending anchor like `$`. If the analysis engine can determine a maximum number of characters for any possible match, and it has such an anchor, then it can simply jump to that distance from the end of the string, and bypass even looking at anything before then.

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;
private Regex _regex = new Regex(@"^abc|^def", RegexOptions.Compiled);

[Benchmark]
public bool IsMatch() => _regex.IsMatch(s_haystack); // Why search _all_ the text?!
```

Method	Runtime	Mean	Ratio
IsMatch	.NET 6.0	867,890.56 ns	1.000
IsMatch	.NET 7.0	33.55 ns	0.000

[dotnet/runtime#67732](https://github.com/dotnet/runtime/pull/67732) is another PR related to improving anchor handling. It's always fun when a bug fix or code simplification refactoring turns into a performance improvement. The PR's primary purpose was to simplify some complicated code that was computing the set of characters that could possibly start a match. It turns out that complication was hiding a logic bug which manifested in it missing some opportunities to report valid starting character classes, the impact of which is that some searches which could have been vectorized weren't. By simplifying the implementation, the bug was fixed, exposing more performance opportunities.

By this point, the engines are able to use `IndexOf(ReadOnlySpan<char>)` to find a substring at the beginning of a pattern. But sometimes the most valuable substring isn't at the beginning, but somewhere in the middle or even at the end. As long as it's at a fixed-offset from the beginning of the pattern, we can search for it, and then just back-off by the offset to the position we should actually try running the match. dotnet/runtime#67907 does exactly that.

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;
private Regex _regex = new Regex(@"looking|feeling", RegexOptions.Compiled);

[Benchmark]
public int Count() => _regex.Matches(s_haystack).Count; // will search for "ing"
```

Method	Runtime	Mean	Ratio
Count	.NET 6.0	444.2 us	1.00
Count	.NET 7.0	122.6 us	0.28

Loops and Backtracking

Loop handling in the compiled and source generated engines has been significantly improved, both with respect to processing them faster and with respect to backtracking less.

With regular greedy loops (e.g. `c*`), there are two directions to be concerned about: how quickly can we consume all the elements that match the loop, and how quickly can we give back elements that might be necessary as part of backtracking for the remainder of the expression to match. And with lazy loops, we're primarily concerned with backtracking, which is the forward direction (since lazy loops consume as part of backtracking rather than giving back as part of backtracking). With PRs dotnet/runtime#63428, dotnet/runtime#68400, dotnet/runtime#64254, and dotnet/runtime#73910, in both the compiler and source generator we now make full use of effectively all of the variants of `IndexOf`, `IndexOfAny`, `LastIndexOf`, `LastIndexOfAny`, `IndexOfAnyExcept`, and `LastIndexOfAnyExcept` in order to speed along these searches. For example, in a pattern like `.*abc`, the forward direction of that loop entails consuming every character until the next newline, which we can optimize with an `IndexOf('\n')`. Then as part of backtracking, rather than giving up one character at a time, we can `LastIndexOf("abc")` in order to find the next viable location that could possibly match the remainder of the pattern. Or for example, in a pattern like `[^a-c]*def`, the loop will initially greedily consume everything other than `'a'`, `'b'`, or `'c'`, so we can use `IndexOfAnyExcept('a', 'b', 'c')` to find the initial end of the loop. And so on. This can yield huge performance gains, and with the source generator, also makes the generated code more idiomatic and easier to understand.

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;
private Regex _regex = new Regex(@"^.*elementary.*$", RegexOptions.Compiled |
RegexOptions.Multiline);

[Benchmark]
public int Count() => _regex.Matches(s_haystack).Count;
```

Method	Runtime	Mean	Ratio
Count	.NET 6.0	3,369.5 us	1.00
Count	.NET 7.0	430.2 us	0.13

Sometimes optimizations are well-intended but slightly miss the mark. dotnet/runtime#63398 fixes such an issue with an optimization introduced in .NET 5; the optimization was valuable but only for a subset of the scenarios it was intended to cover. While `TryFindNextPossibleStartingPosition`'s primary raison d'être is to update the bumpalong position, it's also possible for `TryMatchAtCurrentPosition` to do so. One of the occasions in which it'll do so is when the pattern begins with an upper-unbounded single-character greedy loop. Since processing starts with the loop having fully consumed everything it could possibly match, subsequent trips through the scan loop don't need to reconsider any starting position within that loop; doing so would just be duplicating work done in a previous iteration of the scan loop. And as such, `TryMatchAtCurrentPosition` can update the bumpalong position to the end of the loop. The optimization added in .NET 5 was dutifully doing this, and it did so in a way that fully handled atomic loops. But with greedy loops, the updated position was getting updated every time we backtracked, meaning it started going backwards, when it should have remained at the end of the loop. This PR fixes that, yielding significant savings in the additional covered cases.

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;
private Regex _regex = new Regex(@".*stephen", RegexOptions.Compiled);

[Benchmark]
public int Count() => _regex.Matches(s_haystack).Count;
```

Method	Runtime	Mean	Ratio
Count	.NET 6.0	103,962.8 us	1.000
Count	.NET 7.0	336.9 us	0.003

As mentioned elsewhere, the best optimizations are those that make work entirely vanish rather than just making work faster. dotnet/runtime#68989, dotnet/runtime#63299, and dotnet/runtime#63518 do exactly that by improving the pattern analyzers ability to find and eliminate more unnecessary backtracking, a process the analyzer refers to as "auto-atomicity" (automatically making loops atomic). For example, in the pattern `a*?b`, we have a lazy loop of `'a'`s followed by a `b`. That loop can only match `'a'`s, and `'a'` doesn't overlap with `'b'`. So let's say the input is `"aaaaaaaab"`. The loop is lazy, so we'll start out by trying to match just `'b'`. It won't match, so we'll backtrack into the lazy loop and try to match `"ab"`. It won't match so we'll backtrack into the lazy loop and try to match `"aab"`. And so on, until we've consumed all the `'a'`s such that the rest of the pattern has a chance of matching the rest of the input. That's exactly what an atomic greedy loop does, so we can transform the pattern `a*?b` into `(?>a*)b`, which is much more efficiently processed. In fact, we can see exactly how it's processed just by looking at the source-generated implementation of this pattern:

```
private bool TryMatchAtCurrentPosition(ReadOnlySpan<char> inputSpan)
{
    int pos = base.runtextpos;
    int matchStart = pos;
    ReadOnlySpan<char> slice = inputSpan.Slice(pos);
```

```
    // Match 'a' atomically any number of times.
    {
        int iteration = slice.IndexOfAnyExcept('a');
        if (iteration < 0)
        {
            iteration = slice.Length;
        }

        slice = slice.Slice(iteration);
        pos += iteration;
    }

    // Advance the next matching position.
    if (base.runtextpos < pos)
    {
        base.runtextpos = pos;
    }

    // Match 'b'.
    if (slice.IsEmpty || slice[0] != 'b')
    {
        return false; // The input didn't match.
    }

    // The input matched.
    pos++;
    base.runtextpos = pos;
    base.Capture(0, matchStart, pos);
    return true;
}
```

(Note that those comments aren't ones I added for this blog post; the source generator itself is emitting commented code.)

When a regular expression is input, it's parsed into a tree-based form. The "auto-atomicity" analysis discussed in the previous PR is one form of analysis that walks around this tree looking for opportunities to transform portions of the tree into a behaviorally equivalent alternative that will be more efficient to execute. Several PRs introduced additional such transformations. dotnet/runtime#63695, for example, looks for "empty" and "nothing" nodes in the tree that can be removed. An "empty" node is something that matches the empty string, so for example in the alternation `abc|def||ghi`, the third branch of that alternation is empty. A "nothing" node is something that can't match anything, so for example in the concatenation `abc(?!)def`, that `(?!)` in middle is a negative lookahead around an empty, which can't possibly match anything, as it's saying the expression won't match if it's followed by an empty string, which everything is. These constructs often arise as a result of other transformations rather than being something a developer typically writes by hand, just as there are optimizations in the JIT where you might look at them and say "why on earth is that something a developer would write" but it ends up being a valuable optimization anyways because inlining might transform perfectly reasonable code into something that matches the target pattern. Thus, for example, if you did have `abc(?!)def`, since that concatenation requires the `(?!)` to match in order to be successful, the concatenation itself can simply be replaced by a "nothing." You can see this easily if you try this with the source generator:

```
[GeneratedRegex(@"abc(?!)def")]
```

as it will produce a `Scan` method like this (comment and all):

```
protected override void Scan(ReadOnlySpan<char> inputSpan)
{
    // The pattern never matches anything.
}
```

Another set of transformations was introduced in [dotnet/runtime#59903](), specifically around alternations (which beyond loops are the other source of backtracking). This introduced two main optimizations. First, it enables rewriting alternations into alternations of alternations, e.g. transforming `axy|axz|bxy|bxz` into `ax(?:y|z)|bx(?:y|z)`, which is then further reduced into `ax[yz]|bx[yz]`. This can enable the backtracking engines to more efficiently process alternations due to fewer branches and thus less potential backtracking. The PR also enabled limited reordering of branches in an alternation. Generally branches can't be reordered, as the order can impact exactly what's matched and what's captured, but if the engine can prove there's no effect on ordering, then it's free to reorder. One key place that ordering isn't a factor is if the alternation is atomic due to it being wrapped in an atomic group (and the auto-atomicity analysis will add such groups implicitly in some situations). Reordering the branches then enables other optimizations, like the one previously mentioned from this PR. And then once those optimizations have kicked in, if we're left with an atomic alternation where every branch begins with a different letter, than can enable further optimizations in terms of how the alternation is lowered; this PR teaches the source generator how to emit a `switch` statement, which leads to both more efficient and more readable code. (The detection of whether nodes in the tree are atomic, and other such properties such as performing captures or introducing backtracking, turned out to be valuable enough that [dotnet/runtime#65734]() added dedicated support for this.)

Code generation

The .NET 7 regex implementation has no fewer than four engines: the interpreter (what you get if you don't explicitly choose another engine), the compiler (what you get with `RegexOptions.Compiled`), the non-backtracking engine (what you get with `RegexOptions.NonBacktracking`), and the source generator (what you get with `[GeneratedRegex(...)]`). The interpreter and the non-backtracking engine don't require any kind of code generation; they're both based on creating in-memory data structures that represent how to match input against the pattern. The other two, though, both generate code specific to the pattern; the generated code is code attempting to mimick what you might write if you weren't using `Regex` at all and were instead writing code to perform a similar match directly. The source generator spits out C# that's compiled directly into your assembly, and the compiler spits out IL at run-time via reflection emit. The fact that these are generating code specific to the pattern means there's a ton of opportunity to optimize.

[dotnet/runtime#59186]() provided the initial implementation of the source generator. This was a direct port of the compiler, effectively a line-by-line translation of IL into C#; the result is C# akin to what you'd get if you were to run the generated IL through a decompiler like [ILSpy](). A bunch of PRs then proceeded to iterate on and tweak the source generator, but the biggest improvements came from changes that changed the compiler and the source generator together. Prior to .NET 5, the compiler spit out IL that was very similar to what the interpreter would do. The interpreter is handed a series of instructions that it walks through one by one and interprets, and the compiler, handed that same series of instructions, would just emit the IL for processing each. It had some opportunity for being more efficient, e.g. loop unrolling, but a lot of value was left on the table. In .NET 5, an alternate path was added in support of patterns without backtracking; this code path was based on being handed the parsed node tree rather than being based on the series of instructions, and that higher-level form enabled the compiler to derive more insights about the pattern that it could then use to generate more efficient code. In .NET 7, support for all regex features were incrementally added in, over the course of multiple PRs, in particular [dotnet/runtime#60385](https://github.com/dotnet/runtime/pull/60385) for backtracking single char loops, [dotnet/runtime#61698](https://github.com/dotnet/runtime/pull/61698) for backtracking single char lazy loops, [dotnet/runtime#61784](https://github.com/dotnet/runtime/pull/61784) for other backtracking lazy loops, and [dotnet/runtime#61906](https://github.com/dotnet/runtime/pull/61906) for other backtracking loops as well as back references and conditionals. At that point, the only features missing were support for `RegexOptions.RightToLeft` and lookbehinds (which are implemented in terms of right-to-left), and we decided based on relatively little use of these features that we needn't keep around the old compiler code just to enable them. So, [dotnet/runtime#62318](https://github.com/dotnet/runtime/pull/62318) deleted the old implementation. But, even though these features are relatively rare, it's a lot easier to tell a story that "all patterns are supported" than one that requires special callouts and exceptions, so [dotnet/runtime#66127](https://github.com/dotnet/runtime/pull/66127) and [dotnet/runtime#66280](https://github.com/dotnet/runtime/pull/66280) added full lookbehind and `RightToLeft` support such that there were no takebacks. At this point, both the compiler and source generator now supported everything the compiler previously did, but now with the more modernized code generation. This code generation is in turn what enables many of the optimizations previously discussed, e.g. it provides the opportunity to use APIs like `LastIndexOf` as part of backtracking, which would have been near impossible with the previous approach.

One of the great things about the source generator emitting idiomatic C# is it makes it easy to iterate. Every time you put in a pattern and see what the generator emits, it's like being asked to do a code review of someone else's code, and you very frequently see something "new" worthy of comment, or in this case, improving the generator to address the issue. And so a bunch of PRs were originated based on reviewing what the generator emitted and then tweaking the generator to do better (and since the compiler was effectively entirely rewritten along with the source generator, they maintain the same structure, and it's easy to port improvements from one to the other). For example, [dotnet/runtime#68846](https://github.com/dotnet/runtime/pull/68846) and [dotnet/runtime#69198](https://github.com/dotnet/runtime/pull/69198) tweaked how some comparisons were being performed in order for them to convey enough information to the JIT that it can eliminate some subsequent bounds checking, and [dotnet/runtime#68490](https://github.com/dotnet/runtime/pull/68490) recognized a variety of conditions being emitted that could never happen in some situations observable statically and was able to elide all that code gen. It also became obvious that some patterns didn't need the full expressivity of the scan loop, and a more compact and customized `Scan` implementation could be used. [dotnet/runtime#68560](https://github.com/dotnet/runtime/pull/68560) does that, such that, for example, a simple pattern like `hello` won't emit a loop at all and will instead have a simpler `Scan` implementation like:

```
protected override void Scan(ReadOnlySpan<char> inputSpan)
{
    if (TryFindNextPossibleStartingPosition(inputSpan))
    {
        // The search in TryFindNextPossibleStartingPosition performed the entire match.
        int start = base.runtextpos;
        int end = base.runtextpos = start + 5;
        base.Capture(0, start, end);
    }
}
```

The compiler and source generator were also updated to take advantage of newer features. [dotnet/runtime#63277](), for example, teaches the source generator how to determine if `unsafe` code is allowed, and if it is, it emits a `[SkipLocalsInit]` for the core logic; the matching routine can result in many locals being emitted, and `SkipLocalsInit` can make it cheaper to call the function due to less zero'ing being necessary. Then there's the issue of where the code is generated; we want helper functions (like the `\w` `IsWordChar` helper introduced in [dotnet/runtime#62620]()) that can be shared amongst multiple generated regexes, and we want to be able to share the exact same regex implementation if the same pattern/options/timeout combination are used in multiple places in the same assembly ([dotnet/runtime#66747]()), but doing so then exposes this implementation detail to user code in the same assembly. To still be able to get the perf benefits of such code sharing while avoiding the resulting complications, [dotnet/runtime#66432]() and then [dotnet/runtime#71765]() teaches the source generator to use the new `file-local types` features in C# 11 ([dotnet/roslyn#62375]()).

One last and interesting code generation aspect is in optimizations around character class matching. Matching character classes, whether ones explicitly written by the developer or ones implicitly created by the engine (e.g. as part of finding the set of all characters that can begin the expression), can be one of the more time-consuming aspects of matching; if you imagine having to evaluate this logic for every character in the input, then how many instructions needs to be executed as part of matching a character class directly correlates to how long it takes to perform the overall match. We thus spend some time trying to ensure we generate optimal matching code for as many categories of character classes as possible. [dotnet/runtime#67365](), for example, improved a bunch of cases found to be common in real-world use, like specially-recognizing sets like `[\d\D]`, `[\s\S]`, and `[\w\W]` as meaning "match anything" (just as is the case for `.` in `RegexOptions.Singleline` mode), in which case existing optimizations around the handling of "match anything" can kick in.

```
private static readonly string s_haystack = new string('a', 1_000_000);
private Regex _regex = new Regex(@"([\s\S]*)", RegexOptions.Compiled);

[Benchmark]
public Match Match() => _regex.Match(s_haystack);
```

Method	Runtime	Mean	Ratio
Match	.NET 6.0	1,934,393.69 ns	1.000
Match	.NET 7.0	91.80 ns	0.000

Or [dotnet/runtime#68924](), which taught the source generator how to use all of the new `char` ASCII helper methods, like `char.IsAsciiLetterOrDigit`, as well as some existing helpers it didn't yet know about, in the generated output; for example this:

```
[GeneratedRegex(@"[A-Za-z][A-Z][a-z][0-9][A-Za-z0-9][0-9A-F][0-9a-f][0-9A-Fa-
f]\p{Cc}\p{L}[\p{L}\d]\p{Ll}\p{Lu}\p{N}\p{P}\p{Z}\p{S}")]
```

now produces this in the core matching logic emitted by the source generator:

```
if ((uint)slice.Length < 17 ||
    !char.IsAsciiLetter(slice[0]) || // Match a character in the set [A-Za-z].
    !char.IsAsciiLetterUpper(slice[1]) || // Match a character in the set [A-Z].
    !char.IsAsciiLetterLower(slice[2]) || // Match a character in the set [a-z].
    !char.IsAsciiDigit(slice[3]) || // Match '0' through '9'.
    !char.IsAsciiLetterOrDigit(slice[4]) || // Match a character in the set [0-9A-Za-z].
```

```
    !char.IsAsciiHexDigitUpper(slice[5]) || // Match a character in the set [0-9A-F].
    !char.IsAsciiHexDigitLower(slice[6]) || // Match a character in the set [0-9a-f].
    !char.IsAsciiHexDigit(slice[7]) || // Match a character in the set [0-9A-Fa-f].
    !char.IsControl(slice[8]) || // Match a character in the set [\p{Cc}].
    !char.IsLetter(slice[9]) || // Match a character in the set [\p{L}].
    !char.IsLetterOrDigit(slice[10]) || // Match a character in the set [\p{L}\d].
    !char.IsLower(slice[11]) || // Match a character in the set [\p{Ll}].
    !char.IsUpper(slice[12]) || // Match a character in the set [\p{Lu}].
    !char.IsNumber(slice[13]) || // Match a character in the set [\p{N}].
    !char.IsPunctuation(slice[14]) || // Match a character in the set [\p{P}].
    !char.IsSeparator(slice[15]) || // Match a character in the set [\p{Z}].
    !char.IsSymbol(slice[16])) // Match a character in the set [\p{S}].
{
    return false; // The input didn't match.
}
```

Other changes impacting character class code generation included dotnet/runtime#72328, which improved the handling of character classes that involve character class subtraction; dotnet/runtime#72317 from [@teo-tsirpanis](https://github.com/teo-tsirpanis), which enabled additional cases where the generator could avoid emitting a bitmap lookup; dotnet/runtime#67133, which added a tighter bounds check when it does emit such a lookup table; and dotnet/runtime#61562, which enables better normalization of character classes in the engine's internal representation, thus leading to downstream optimizations better recognizing more character classes.

Finally, with all of these improvements to `Regex`, a multitude of PRs fixed up regexes being used across dotnet/runtime, in various ways. dotnet/runtime#66142, dotnet/runtime#66179 from [@Clockwork-Muse](https://github.com/Clockwork-Muse), and dotnet/runtime#62325 from [@Clockwork-Muse](https://github.com/Clockwork-Muse) all converted `Regex` usage over to using `[GeneratedRegex(...)]`. dotnet/runtime#68961 optimized other usage in various ways. The PR replaced several `regex.Matches(...).Success` calls with `IsMatch(...)`, as using `IsMatch` has less overhead due to not needing to construct a `Match` instance and due to being able to avoid more expensive phases in the non-backtracking engine to compute exact bounds and capture information. The PR also replaced some `Match`/`Match.MoveNext` usage with `EnumerateMatches`, in order to avoid needing `Match` object allocations. The PR also entirely removed at least one regex usage that was just as doable as a cheaper `IndexOf`. dotnet/runtime#68766 also removed a use of `RegexOptions.CultureInvariant`. Specifying `CultureInvariant` changes the behavior of `IgnoreCase` by alternating which casing tables are employed; if `IgnoreCase` isn't specified and there's no inline case-insensitivity options (`(?i)`), then specifying `CultureInvariant` is a nop. But a potentially expensive one. For any code that's size conscious, the `Regex` implementation is structured in a way as to try to make it as trimmmer friendly as possible. If you only ever do `new Regex(pattern)`, we'd really like to be able to statically determine that the compiler and non-backtracking implementations aren't needed such that the trimmer can remove it without having a visible and meaningful negative impact. However, the trimmer analysis isn't yet sophisticated enough to see exactly which options are used and only keep the additional engines linked in if `RegexOptions.Compiled` or `RegexOptions.NonBacktracking` is used; instead, *any* use of an overload that takes a `RegexOptions` will result in that code continuing to be referenced. By getting rid of the options, we increase the chances that no code in the app is using this constructor, which would in turn enable this constructor, the compiler, and the non-backtracking implementation to be trimmed away.

CHAPTER 12

Collections

`System.Collections` hasn't seen as much investment in .NET 7 as it has in previous releases, though many of the lower-level improvements have a trickle-up effect into collections as well. For example, `Dictionary<,>`'s code hasn't changed between .NET 6 and .NET 7, but even so, this benchmark focused on dictionary lookups:

```
private Dictionary<int, int> _dictionary = Enumerable.Range(0, 10_000).ToDictionary(i =>
i);

[Benchmark]
public int Sum()
{
    Dictionary<int, int> dictionary = _dictionary;
    int sum = 0;

    for (int i = 0; i < 10_000; i++)
    {
        if (dictionary.TryGetValue(i, out int value))
        {
            sum += value;
        }
    }

    return sum;
}
```

shows a measurable improvement in throughput between .NET 6 and .NET 7:

Method	Runtime	Mean	Ratio	Code Size
Sum	.NET 6.0	51.18 us	1.00	431 B
Sum	.NET 7.0	43.44 us	0.85	413 B

Beyond that, there have been explicit improvements elsewhere in collections. `ImmutableArray<T>`, for example. As a reminder, `ImmutableArray<T>` is a very thin struct-based wrapper around a `T[]` that hides the mutability of `T[]`; unless you're using unsafe code, neither the length nor the shallow contents of an `ImmutableArray<T>` will ever change (by shallow, I mean the data stored directly in that array can't be mutated, but if there are mutable reference types stored in the array, those instances themselves may still have their data mutated). As a result, `ImmutableArray<T>` also has an associated "builder" type, which does support mutation: you create the builder, populate it, and then transfer that contents to an `ImmutableArray<T>` which is frozen forevermore. In [dotnet/runtime#70850](https://github.com/dotnet/runtime/pull/70850) from [@grbell-ms](https://github.com/grbell-ms), the builder's `Sort` method is changed to use a span, which in turn avoids an `IComparer<T>` allocation and a `Comparison<T>`

allocation, while also speeding up the sort itself by removing several layers of indirection from every comparison.

```
private ImmutableArray<int>.Builder _builder = ImmutableArray.CreateBuilder<int>();

[GlobalSetup]
public void Setup()
{
    _builder.AddRange(Enumerable.Range(0, 1_000));
}

[Benchmark]
public void Sort()
{
    _builder.Sort((left, right) => right.CompareTo(left));
    _builder.Sort((left, right) => left.CompareTo(right));
}
```

Method	Runtime	Mean	Ratio
Sort	.NET 6.0	86.28 us	1.00
Sort	.NET 7.0	67.17 us	0.78

dotnet/runtime#61196 from [@lateapexearlyspeed](https://github.com/lateapexearlyspeed) brings `ImmutableArray<T>` into the span-based era, adding around 10 new methods to `ImmutableArray<T>` that interoperate with `Span<T>` and `ReadOnlySpan<T>`. These are valuable from a performance perspective because it means if you have your data in a span, you can get it into an `ImmutableArray<T>` without incurring additional allocations beyond the one the `ImmutableArray<T>` itself will create. dotnet/runtime#66550 from [@RaymondHuy](https://github.com/RaymondHuy) also adds a bunch of new methods to the immutable collection builders, which provide efficient implementations for operations like replacing elements and adding, inserting, and removing ranges.

`SortedSet<T>` also saw some improvements in .NET 7. For example, `SortedSet<T>` internally uses a red/black tree as its internal data structure, and it uses a `Log2` operation to determine the maximum depth the tree could be for a given node count. Previously, that operation was implemented as a loop. But thanks to dotnet/runtime#58793 from [@teo-tsirpanis](https://github.com/teo-tsirpanis) that implementation is now simply a call to `BitOperations.Log2`, which is in turn implemented trivially in terms of one of multiple hardware intrinsics if they're supported (e.g. `Lzcnt.LeadingZeroCount`, `ArmBase.LeadingZeroCount`, `X86Base.BitScanReverse`). And dotnet/runtime#56561 from [@johnthcall](https://github.com/johnthcall) improves `SortedSet<T>` copy performance by streamlining how the iteration through the nodes in the tree is handled.

```
[Params(100)]
public int Count { get; set; }

private static SortedSet<string> _set;

[GlobalSetup]
public void GlobalSetup()
{
    _set = new SortedSet<string>(StringComparer.OrdinalIgnoreCase);
    for (int i = 0; i < Count; i++)
    {
        _set.Add(Guid.NewGuid().ToString());
```

```
    }
}

[Benchmark]
public SortedSet<string> SortedSetCopy()
{
    return new SortedSet<string>(_set, StringComparer.OrdinalIgnoreCase);
}
```

Method	Runtime	Mean	Ratio
SortedSetCopy	.NET 6.0	2.397 us	1.00
SortedSetCopy	.NET 7.0	2.090 us	0.87

One last PR to look at in collections: dotnet/runtime#67923. `ConditionalWeakTable<TKey, TValue>` is a collection most developers haven't used, but when you need it, you need it. It's used primarily for two purposes: to associate additional state with some object, and to maintain a weak collection of objects. Essentially, it's a thread-safe dictionary that doesn't maintain strong references to anything it stores but ensures that the value associated with a key will remain rooted as long as the associated key is rooted. It exposes many of the same APIs as `ConcurrentDictionary<,>`, but for adding items to the collection, it's historically only had an `Add` method. That means if the design of the consuming code entailed trying to use the collection as a set, where duplicates were common, it would also be common to experience exceptions when trying to `Add` an item that already existed in the collection. Now in .NET 7, it has a `TryAdd` method, which enables such usage without potentially incurring the costs of such exceptions (and without needing to add `try/catch` blocks to defend against them).

CHAPTER 13

LINQ

Let's move on to Language-Integrated Query (LINQ). LINQ is a productivity feature that practically every .NET developer uses. It enables otherwise complicated operations to be trivially expressed, whether via language-integrated query comprehension syntax or via direct use of methods on `System.Linq.Enumerable`. That productivity and expressivity, however, comes at a bit of an overhead cost. In the vast majority of situations, those costs (such as delegate and closure allocations, delegate invocations, use of interface methods on arbitrary enumerables vs direct access to indexers and `Length`/`Count` properties, etc.) don't have a significant impact, but for really hot paths, they can and do show up in a meaningful way. This leads some folks to declare LINQ as being broadly off-limits in their codebases. From my perspective, that's misguided; LINQ is extremely useful and has its place. In .NET itself, we use LINQ, we're just practical and thoughtful about where, avoiding it in code paths we've optimized to be lightweight and fast due to expectations that such code paths could matter to consumers. And as such, while LINQ itself may not perform as fast as a hand-rolled solution, we still care a lot about the performance of LINQ's implementation, so that it can be used in more and more places, and so that where it's used there's as little overhead as possible. There are also differences between operations in LINQ; with over 200 overloads providing various kinds of functionality, some of these overloads benefit from more performance tuning than do others, based on their expected usage.

dotnet/runtime#64470 is the result of analyzing various real-world code bases for use of `Enumerable.Min` and `Enumerable.Max`, and seeing that it's very common to use these with arrays, often ones that are quite large. This PR updates the `Min<T>(IEnumerable<T>)` and `Max<T>(IEnumerable<T>)` overloads when the input is an `int[]` or `long[]` to vectorize the processing, using `Vector<T>`. The net effect of this is significantly faster execution time for larger arrays, but still improved performance even for short arrays (because the implementation is now able to access the array directly rather than going through the enumerable, leading to less allocation and interface dispatch and more applicable optimizations like inlining).

```
[Params(4, 1024)]
public int Length { get; set; }

private IEnumerable<int> _source;

[GlobalSetup]
public void Setup() => _source = Enumerable.Range(1, Length).ToArray();

[Benchmark]
public int Min() => _source.Min();

[Benchmark]
public int Max() => _source.Max();
```

Method	Runtime	Length	Mean	Ratio	Allocated	Alloc Ratio
Min	.NET 6.0	4	26.167 ns	1.00	32 B	1.00
Min	.NET 7.0	4	4.788 ns	0.18	-	0.00
Max	.NET 6.0	4	25.236 ns	1.00	32 B	1.00
Max	.NET 7.0	4	4.234 ns	0.17	-	0.00
Min	.NET 6.0	1024	3,987.102 ns	1.00	32 B	1.00
Min	.NET 7.0	1024	101.830 ns	0.03	-	0.00
Max	.NET 6.0	1024	3,798.069 ns	1.00	32 B	1.00
Max	.NET 7.0	1024	100.279 ns	0.03	-	0.00

One of the more interesting aspects of the PR, however, is one line that's meant to help with the non-array cases. In performance optimization, and in particular when adding "fast paths" to better handle certain cases, there's almost always a winner and a loser: the winner is the case the optimization is intended to help, and the loser is every other case that's penalized by whatever checks are necessary to determine whether to take the improved path. An optimization that special-cases arrays might normally look like:

```
if (source is int[] array)
{
    ProcessArray(array);
}
else
{
    ProcessEnumerable(source);
}
```

However, if you look at the PR, you'll see the `if` condition is actually:

```
if (source.GetType() == typeof(int[]))
```

How come? Well at this point in the code flow, we know that source isn't null, so we don't need the extra null check that `is` will bring. However, that's minor compared to the real impact here, that of support for array covariance. It might surprise you to learn that there are types beyond `int[]` that will satisfy a `source is int` check… try running `Console.WriteLine((object)new uint[42] is int[]);`, and you'll find it prints out `True`. (This is also a rare case where the .NET runtime and C# the language disagree on aspects of the type system. If you change that `Console.WriteLine((object)new uint[42] is int[]);` to instead be `Console.WriteLine(new uint[42] is int[]);`, i.e. remove the `(object)` cast, you'll find it starts printing out `False` instead of `True`. That's because the C# compiler believes it's impossible for a `uint[]` to ever be an `int[]`, and thus optimizes the check away entirely to be a constant `false`.) Thus the runtime is having to do more work as part of the type check than just a simple comparison against the known type identity of `int[]`. We can see this by looking at the assembly generated for these two methods (the latter assumes we've already null-checked the input, which is the case in these LINQ methods):

```
public IEnumerable<object> Inputs { get; } = new[] { new object() };

[Benchmark]
[ArgumentsSource(nameof(Inputs))]
public bool M1(object o) => o is int[];

[Benchmark]
[ArgumentsSource(nameof(Inputs))]
public bool M2(object o) => o.GetType() == typeof(int[]);
```

This results in:

```
; Program.M1(System.Object)
       sub       rsp,28
       mov       rcx,offset MT_System.Int32[]
       call      qword ptr
[System.Runtime.CompilerServices.CastHelpers.IsInstanceOfAny(Void*, System.Object)]
       test      rax,rax
       setne     al
       movzx     eax,al
       add       rsp,28
       ret
; Total bytes of code 34
```

```
; Program.M2(System.Object)
       mov       rax,offset MT_System.Int32[]
       cmp       [rdx],rax
       sete      al
       movzx     eax,al
       ret
; Total bytes of code 20
```

Note the former involves a method call to the JIT's `CastHelpers.IsInstanceOfAny` helper method, and that it's not inlined. That in turn impacts performance:

```
private IEnumerable<int> _source = (int[])(object)new uint[42];

[Benchmark(Baseline = true)]
public bool WithIs() => _source is int[];

[Benchmark]
public bool WithTypeCheck() => _source.GetType() == typeof(int[]);
```

Method	Mean	Ratio	Code Size
WithIs	1.9246 ns	1.000	215 B
WithTypeCheck	0.0013 ns	0.001	24 B

Of course, these two operations aren't semantically equivalent, so if this was for something that required the semantics of the former, we couldn't use the latter. But in the case of this LINQ performance optimization, we can choose to only optimize the `int[]` case, forego the super rare case of the `int[]` actually being a `uint[]` (or e.g. `DayOfWeek[]`), and minimize the performance penalty of the optimization for `IEnumerable<int>` inputs other than `int[]` to just a few quick instructions.

This improvement was built upon further in [dotnet/runtime#64624](https://github.com/dotnet/runtime/pull/64624), which expands the input types supported and the operations that take advantage. First, it introduced a private helper for extracting a `ReadOnlySpan<T>` from certain types of `IEnumerable<T>` inputs, namely today those inputs that are actually either a `T[]` or a `List<T>`; as with the previous PR, it uses the `GetType() == typeof(T[])` form to avoid significantly penalizing other inputs. Both of these types enable extracting a `ReadOnlySpan<T>` for the actual storage, in the case of `T[]` via a cast and in the case of `List<T>` via the `CollectionsMarshal.AsSpan` method that was introduced in .NET 5. Once we have that span, we can do a few interesting things. This PR:

- Expands the previous `Min<T>(IEnumerable<T>)` and `Max<T>(IEnumerable<T>)` optimizations to not only apply to `int[]` and `long[]` but also to `List<int>` and `List<long>`.
- Uses direct span access for `Average<T>(IEnumerable<T>)` and `Sum<T>(IEnumerable<T>)` for `T` being `int`, `long`, `float`, `double`, or `decimal`, all for arrays and lists.
- Similarly uses direct span access for `Min<T>(IEnumerable<T>)` and `Max<T>(IEnumerable<T>)` for `T` being `float`, `double`, and `decimal`.
- Vectorizes `Average<int>(IEnumerable<int>)` for arrays and lists

The effect of that is evident in microbenchmarks, e.g.

```
private static float[] CreateRandom()
{
    var r = new Random(42);
    var results = new float[10_000];
    for (int i = 0; i < results.Length; i++)
    {
        results[i] = (float)r.NextDouble();
    }
    return results;
}

private IEnumerable<float> _floats = CreateRandom();

[Benchmark]
public float Sum() => _floats.Sum();

[Benchmark]
public float Average() => _floats.Average();

[Benchmark]
public float Min() => _floats.Min();

[Benchmark]
public float Max() => _floats.Max();
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Sum	.NET 6.0	39.067 us	1.00	32 B	1.00
Sum	.NET 7.0	14.349 us	0.37	-	0.00
Average	.NET 6.0	41.232 us	1.00	32 B	1.00

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Average	.NET 7.0	14.378 us	0.35	-	0.00
Min	.NET 6.0	45.522 us	1.00	32 B	1.00
Min	.NET 7.0	9.668 us	0.21	-	0.00
Max	.NET 6.0	41.178 us	1.00	32 B	1.00
Max	.NET 7.0	9.210 us	0.22	-	0.00

The previous LINQ PRs were examples from making existing operations faster. But sometimes performance improvements come about from new APIs that can be used in place of previous ones in certain situations to further improve performance. One such example of that comes from new APIs introduced in [dotnet/runtime#70525](https://github.com/dotnet/runtime/pull/70525) from [@deeprobin](https://github.com/deeprobin) which were then improved in [dotnet/runtime#71564](https://github.com/dotnet/runtime/pull/71564). One of the most popular methods in LINQ is `Enumerable.OrderBy` (and its inverse `OrderByDescending`), which enables creating a sorted copy of the input enumerable. To do so, the caller passes a `Func<TSource,TKey>` predicate to `OrderBy` which `OrderBy` uses to extract the comparison key for each item. However, it's relatively common to want to sort items with themselves as the keys; this is, after all, the default for methods like `Array.Sort`, and in such cases callers of `OrderBy` end up passing in an identity function, e.g. `OrderBy(x => x)`. To eliminate that cruft, .NET 7 introduces the new `Order` and `OrderDescending` methods, which, in the spirit of pairs like `Distinct` and `DistinctBy`, perform that same sorting operation, just with an implicit `x => x` done on behalf of the caller. But beyond performance, a nice benefit of this is the implementation then knows that the keys will all be the same as the inputs, and it no longer needs to invoke the callback for each item to retrieve its key nor allocate a new array to store those keys. Thus if you find yourself using LINQ and reaching for `OrderBy(x => x)`, consider instead using `Order()` and reaping the (primarily allocation) benefits:

```
[Params(1024)]
public int Length { get; set; }

private int[] _arr;

[GlobalSetup]
public void Setup() => _arr = Enumerable.Range(1, Length).Reverse().ToArray();

[Benchmark(Baseline = true)]
public void OrderBy()
{
    foreach (int _ in _arr.OrderBy(x => x)) { }
}

[Benchmark]
public void Order()
{
    foreach (int _ in _arr.Order()) { }
}
```

Method	Length	Mean	Ratio	Allocated	Alloc Ratio
OrderBy	1024	68.74 us	1.00	12.3 KB	1.00
Order	1024	66.24 us	0.96	8.28 KB	0.67

File I/O

.NET 6 saw some huge file I/O improvements, in particular a complete rewrite of `FileStream`. While .NET 7 doesn't have any single changes on that scale, it does have a significant number of improvements that measurably "move the needle," and in variety of ways.

One form of performance improvement that also masquerades as a reliability improvement is increasing responsiveness to cancellation requests. The faster something can be canceled, the sooner the system is able to give back valuable resources in use, and the sooner things waiting for that operation to complete are able to be unblocked. There have been several improvements of this ilk in .NET 7.

In some cases, it comes from adding cancelable overloads where things weren't previously cancelable at all. That's the case for [dotnet/runtime#61898](https://github.com/dotnet/runtime/pull/61898) from [@bgrainger](https://github.com/bgrainger), which added new cancelable overloads of `TextReader.ReadLineAsync` and `TextReader.ReadToEndAsync`, and that includes overrides of these methods on `StreamReader` and `StringReader`; [dotnet/runtime#64301](https://github.com/dotnet/runtime/pull/64301) from [@bgrainger](https://github.com/bgrainger) then overrode these methods (and others missing overrides) on the `NullStreamReader` type returned from `TextReader.Null` and `StreamReader.Null` (interestingly, these were defined as two different types, unnecessarily, and so this PR also unified on just having both use the `StreamReader` variant, as it satisfies the required types of both). You can see this put to good use in [dotnet/runtime#66492](https://github.com/dotnet/runtime/pull/66492) from [@lateapexearlyspeed](https://github.com/lateapexearlyspeed), which adds a new `File.ReadLinesAsync` method. This produces an `IAsyncEnumerable<string>` of the lines in the file, is based on a simple loop around the new `StreamReader.ReadLineAsync` overload, and is thus itself fully cancelable.

From my perspective, though, a more interesting form of this is when an existing overload is purportedly cancelable but isn't actually. For example, the base `Stream.ReadAsync` method just wraps the `Stream.BeginRead`/`EndRead` methods, which aren't cancelable, so if a `Stream`-derived type doesn't override `ReadAsync`, attempts to cancel a call to its `ReadAsync` will be minimally effective. It does an up-front check for cancellation, such that if cancellation was requested prior to the call being made, it will be immediately canceled, but after that check the supplied `CancellationToken` is effectively ignored. Over time we've tried to stamp out all remaining such cases, but a few stragglers have remained. One pernicious case has been with pipes. For this discussion, there are two relevant kinds of pipes, anonymous and named, which are represented in .NET as pairs of streams: `AnonymousPipeClientStream`/`AnonymousPipeServerStream` and `NamedPipeClientStream`/`NamedPipeServerStream`. Also, on Windows, the OS makes a distinction between handles opened for synchronous I/O from handles opened for overlapped I/O (aka asynchronous I/O), and this is reflected in the .NET API: you can open a named pipe for synchronous or overlapped I/O based on the `PipeOptions.Asynchronous` option specified at construction. And, on

Unix, named pipes, contrary to their naming, are actually implemented on top of Unix domain sockets. Now some history:

- .NET Framework 4.8: No cancellation support. The pipe `Stream`-derived types didn't even override `ReadAsync` or `WriteAsync`, so all they got was the default up-front check for cancellation and then the token was ignored.
- .NET Core 1.0: On Windows, with a named pipe opened for asynchronous I/O, cancellation was fully supported. The implementation would register with the `CancellationToken`, and upon a cancellation request, would use `CancelIoEx` for the `NativeOverlapped*` associated with the asynchronous operation. On Unix, with named pipes implemented in terms of sockets, if the pipe was opened with `PipeOptions.Asynchronous`, the implementation would simulate cancellation via polling: rather than simply issuing the `Socket.ReceiveAsync`/`Socket.SendAsync` (which wasn't cancelable at the time), it would queue a work item to the `ThreadPool`, and that work item would run a polling loop, making `Socket.Poll` calls with a small timeout, checking the token, and then looping around to do it again until either the `Poll` indicated the operation would succeed or cancellation was requested. On both Windows and Unix, other than a named pipe opened with `Asynchronous`, after the operation was initated, cancellation was a nop.
- .NET Core 2.1: On Unix, the implementation was improved to avoid the polling loop, but it still lacked a truly cancelable `Socket.ReceiveAsync`/`Socket.SendAsync`. Instead, by this point `Socket.ReceiveAsync` supported zero-byte reads, where a caller could pass a zero-length buffer to `ReceiveAsync` and use that as notification for data being available to consume without actually consuming it. The Unix implementation for asynchronous named pipe streams then changed to issue zero-byte reads, and would `await` a `Task.WhenAny` of both that operation's task and a task that would be completed when cancellation was requested. Better, but still far from ideal.
- .NET Core 3.0: On Unix, `Socket` got truly cancelable `ReceiveAsync` and `SendAsync` methods, which asynchronous named pipes were updated to utilize. At this point, the Windows and Unix implementations were effectively on par with regards to cancellation; both good for asynchronous named pipes, and just posing for everything else.
- .NET 5: On Unix, `SafeSocketHandle` was exposed and it became possible to create a `Socket` for an arbitrary supplied `SafeSocketHandle`, which enabled creating a `Socket` that actually referred to an anonymous pipe. This in tern enabled every `PipeStream` on Unix to be implemented in terms of `Socket`, which enabled `ReceiveAsync`/`SendAsync` to be fully cancelable for both anonymous and named pipes, regardless of how they were opened.

So by .NET 5, the problem was addressed on Unix, but still an issue on Windows. Until now. In .NET 7, we've made the rest of the operations fully cancelable on Windows as well, thanks to dotnet/runtime#72503 (and a subsequent tweak in dotnet/runtime#72612). Windows doesn't support overlapped I/O for anonymous pipes today, so for anonymous pipes and for named pipes opened for synchronous I/O, the Windows implementation would just delegate to the base `Stream` implementation, which would queue a work item to the `ThreadPool` to invoke the synchronous counterpart, just on another thread. Instead, the implementations now queue that work item, but instead of just calling the synchronous method, it does some pre- and post- work that registers for cancellation, passing in the thread ID of the thread that's about to perform the I/O. If cancellation is requested, the implementation then uses `CancelSynchronousIo` to interrupt it. There's a race condition here, in that the moment the thread registers for cancellation, cancellation could be requested, such that `CancelSynchronousIo` could be called before the operation is actually initiated.

So, there's a small spin loop employed, where if cancellation is requested between the time registration occurs and the time the synchronous I/O is actually performed, the cancellation thread will spin until the I/O is initiated, but this condition is expected to be exceedingly rare. There's also a race condition on the other side, that of `CancelSynchronousIo` being requested after the I/O has already completed; to address that race, the implementation relies on the guarantees made by `CancellationTokenRegistration.Dispose`, which promises that the associated callback will either never be invoked or will already have fully completed executing by the time `Dispose` returns. Not only does this implementation complete the puzzle such that all asynchronous read/write operations on both anonymous and named pipes on both Windows and Unix are cancelable, it also actually improves normal throughput.

```
private Stream _server;
private Stream _client;
private byte[] _buffer = new byte[1];
private CancellationTokenSource _cts = new CancellationTokenSource();

[Params(false, true)]
public bool Cancelable { get; set; }

[Params(false, true)]
public bool Named { get; set; }

[GlobalSetup]
public void Setup()
{
    if (Named)
    {
        string name = Guid.NewGuid().ToString("N");
        var server = new NamedPipeServerStream(name, PipeDirection.Out);
        var client = new NamedPipeClientStream(".", name, PipeDirection.In);
        Task.WaitAll(server.WaitForConnectionAsync(), client.ConnectAsync());
        _server = server;
        _client = client;
    }
    else
    {
        var server = new AnonymousPipeServerStream(PipeDirection.Out);
        var client = new AnonymousPipeClientStream(PipeDirection.In,
server.ClientSafePipeHandle);
        _server = server;
        _client = client;
    }
}

[GlobalCleanup]
public void Cleanup()
{
    _server.Dispose();
    _client.Dispose();
}

[Benchmark(OperationsPerInvoke = 1000)]
public async Task ReadWriteAsync()
{
    CancellationToken ct = Cancelable ? _cts.Token : default;
    for (int i = 0; i < 1000; i++)
    {
```

```
        ValueTask<int> read = _client.ReadAsync(_buffer, ct);
        await _server.WriteAsync(_buffer, ct);
        await read;
    }
}
```

Method	Runtime	Cancelable	Named	Mean	Ratio	Allocated	Alloc Ratio
ReadWriteAsync	.NET 6.0	False	False	22.08 us	1.00	400 B	1.00
ReadWriteAsync	.NET 7.0	False	False	12.61 us	0.76	192 B	0.48
ReadWriteAsync	.NET 6.0	False	True	38.45 us	1.00	400 B	1.00
ReadWriteAsync	.NET 7.0	False	True	32.16 us	0.84	220 B	0.55
ReadWriteAsync	.NET 6.0	True	False	27.11 us	1.00	400 B	1.00
ReadWriteAsync	.NET 7.0	True	False	13.29 us	0.52	193 B	0.48
ReadWriteAsync	.NET 6.0	True	True	38.57 us	1.00	400 B	1.00
ReadWriteAsync	.NET 7.0	True	True	33.07 us	0.86	214 B	0.54

The rest of the performance-focused changes around I/O in .NET 7 were primarily focused on one of two things: reducing syscalls, and reducing allocation.

Several PRs went into reducing syscalls on Unix as part of copying files, e.g. `File.Copy` and `FileInfo.CopyTo`. dotnet/runtime#59695 from [@tmds](https://github.com/tmds) reduced overheads in several ways. The code had been performing a `stat` call in order to determine up front whether the source was actually a directory, in which case the operation would error out. Instead, the PR simply tries to open the source file, which it would need to do anyway for the copy operation, and then it only performs that `stat` if opening the file fails. If opening the file succeeds, the code was already performing an `fstat` to gather data on the file, such as whether it was seekable; with this change, it now also extracts from the results of that single `fstat` the source file size, which it then threads through to the core copy routine, which itself is then able to avoid an `fstat` syscall it had been performing in order to get the size. Saving those syscalls is great, in particular for very small files where the overhead of setting up the copy can actually be more expensive than the actual copy of the bytes. But the biggest benefit of this PR is that it takes advantage of `IOCTL-FICLONERANGE` on Linux. Some Linux file systems, like XFS and Btrfs, support “copy-on-write,” which means that rather than copying all of the data to a new file, the file system simply notes that there are two different files pointing to the same data, sharing the underlying storage. This makes the “copy” super fast, since nothing actually needs to be copied and instead the file system just needs to update some bookkeeping; plus, less space is consumed on disk, since there’s just a single store of the data. The file system then only needs to actually copy data that’s overwritten in one of the files. This PR uses `ioctl` and `FICLONE` to perform the copy as copy-on-write if the source and destination file system are the same and the file system supports the operation. In a similar vein, dotnet/runtime#64264 from [@tmds](https://github.com/tmds) further improves `File.Copy`/`FileInfo.CopyTo` by utilizing `copy_file_range` on Linux if it’s supported (and only if it’s a new enough kernel that it addresses some issues the function had in previous releases). Unlike a typical read/write loop that reads the data from the source and then writes it to the destination, `copy_file_range` is implemented to stay entirely in kernel mode, without having to transition to user space for each read and write.

Another example of avoiding syscalls comes for the `File.WriteXx` and `File.AppendXx` methods when on Unix. The implementation of these methods opens a `FileStream` or a `SafeFileHandle` directly, and it was specifying `FileOptions.SequentialScan`. `SequentialScan` is primarily relevant for reading data from a file, and hints to OS caching to expect data to be read from the file sequentially rather than randomly. However, these write/append methods don't read, they only write, and the implementation of `FileOptions.SequentialScan` on Unix requires an additional syscall via `posix_fadvise` (passing in `POSIX_FADV_SEQUENTIAL`); thus, we're paying for a syscall and not benefiting from it. This situation is akin to the famous Henny Youngman joke: "The patient says, 'Doctor, it hurts when I do this'; the doctor says, 'Then don't do that!'." Here, too, the answer is "don't do that," and so [dotnet/runtime#59247](https://github.com/dotnet/runtime/pull/59247) from [@tmds](https://github.com/tmds) simply stops passing `SequentialScan` in places where it won't help but may hurt.

Directory handling has seen reduced syscalls across the directory lifecycle, especially on Unix. [dotnet/runtime#58799](https://github.com/dotnet/runtime/pull/58799) from [@tmds](https://github.com/tmds) speeds up directory creation on Unix. Previously, the implementation of directory creation would first check to see if the directory already existed, which involves a syscall. In the expected minority case where it already existed the code could early exit out. But in the expected more common case where the directory didn't exist, it would then parse the file path to find all of the directories in it, walk up the directory list until it found one that did exist, and then try to create all of the subdirectories back down through the target one. However, the expected most common case is the parent directories already exist and the child directory doesn't, in which case we're still paying for all that parsing when we could have just created the target directory. This PR addresses that by changing the up-front existence check to instead simply try to `mkdir` the target directory; if it succeeds, great, we're done, and if it fails, the error code from the failure can be used instead of the existence check to know whether `mkdir` failed because it had no work to do. [dotnet/runtime#61777](https://github.com/dotnet/runtime/pull/61777) then takes this a step further and avoids string allocations while creating directories by using stack memory for the paths temporarily needed to pass to `mkdir`.

[dotnet/runtime#63675](https://github.com/dotnet/runtime/pull/63675) then improves the performance of moving directories, on both Unix and Windows, removing several syscalls. The shared code for `Directory.Move` and `DirectorInfo.MoveTo` was doing explicit directory existence checks for the source and destination locations, but on Windows the Win32 API called to perform the move does such checks itself, so they're not needed preemptively. On Unix, we can similarly avoid the existence check for the source directory, as the `rename` function called will similarly simply fail if the source doesn't exist (with an appropriate error that let's us deduce what went wrong so the right exception can be thrown), and for the destination, the code had been issuing separate existence checks for whether the destination existed as a directory or as a file, but a single `stat` call suffices for both.

```
private string _path1;
private string _path2;

[GlobalSetup]
public void Setup()
{
    _path1 = Path.GetTempFileName();
    _path2 = Path.GetTempFileName();
```

```
    File.Delete(_path1);
    File.Delete(_path2);
    Directory.CreateDirectory(_path1);
}

[Benchmark]
public void Move()
{
    Directory.Move(_path1, _path2);
    Directory.Move(_path2, _path1);
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Move	.NET 6.0	31.70 us	1.00	256 B	1.00
Move	.NET 7.0	26.31 us	0.83	-	0.00

And then also on Unix, [dotnet/runtime#59520]() from [@tmds](https://github.com/tmds) improves directory deletion, and in particular recursive deletion (deleting a directory and everything it contains and everything they contain and so on), by utilizing the information already provided by the file system enumeration to avoid a secondary existence check.

Syscalls were also reduced as part of support for memory-mapped files. [dotnet/runtime#63754]() takes advantage of special-casing to do so while opening a `MemoryMappedFile`. When `MemoryMappedFile.CreateFromFile` was called, one of the first things it would do is call `File.Exists` to determine whether the specified file already exists; that's because later in the method as part of dealing with errors and exceptions, the implementation needs to know whether to delete the file that might then exist; the implementation constructs a `FileStream`, and doing might will the specified file into existence. However, that only happens for some `FileMode` values, which is configurable via an argument passed by callers of `CreateFromFile`. The common and default value of `FileMode` is `FileMode.Open`, which requires that the file exist such that constructing the `FileStream` will throw if it doesn't. That means we only actually need to call `File.Exists` if the `FileMode` is something other than `Open` or `CreateNew`, which means we can trivially avoid the extra system call in the majority case. [dotnet/runtime#63790]() also helps here, in two ways. First, throughout the `CreateFromFile` operation, the implementation might access the `FileStream`'s `Length` multiple times, but each call results in a syscall to read the underlying length of the file. We can instead read it once and use that one value for all of the various checks performed. Second, .NET 6 introduced the `File.OpenHandle` method which enables opening a file handle / file descriptor directly into a `SafeFileHandle`, rather than having to go through `FileStream` to do so. The use of the `FileStream` in `MemoryMappedFile` is actually quite minimal, and so it makes sense to just use the `SafeFileHandle` directly rather than also constructing the superfluous `FileStream` and its supporting state. This helps to reduce allocations.

Finally, there's [dotnet/runtime#63794](), which recognizes that a `MemoryMappedViewAccessor` or `MemoryMappedViewStream` opened for read-only access can't have been written to. Sounds obvious, but the practical implication of this is that closing either needn't bother flushing, since that view couldn't have changed any data in the implementation, and flushing a view can be relatively expensive, especially for larger views. Thus, a simple change to avoid flushing if the view isn't writable can yield a measurable improvement to `MemoryMappedViewAccessor`/`MemoryMappedviewStream`'s `Dispose`.

```
private string _path;

[GlobalSetup]
public void Setup()
{
    _path = Path.GetTempFileName();
    File.WriteAllBytes(_path, Enumerable.Range(0, 10_000_000).Select(i =>
(byte)i).ToArray());
}

[GlobalCleanup]
public void Cleanup()
{
    File.Delete(_path);
}

[Benchmark]
public void MMF()
{
    using var mmf = MemoryMappedFile.CreateFromFile(_path, FileMode.Open, null);
    using var s = mmf.CreateViewStream(0, 10_000_000, MemoryMappedFileAccess.Read);
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
MMF	.NET 6.0	315.7 us	1.00	488 B	1.00
MMF	.NET 7.0	227.1 us	0.68	336 B	0.69

Beyond system calls, there have also been a plethora of improvements around reducing allocation. One such change is dotnet/runtime#58167, which improved the performance of the commonly-used `File.WriteAllText{Async}` and `File.AppendAllText{Async}` methods. The PR recognizes two things: one, that these operations are common enough that it's worth avoiding the small-but-measurable overhead of going through a `FileStream` and instead just going directly to the underlying `SafeFileHandle`, and, two, that since the methods are passed the entirety of the payload to output, the implementation can use that knowledge (in particular for length) to do better than the `StreamWriter` that was previously employed. In doing so, the implementation avoids the overheads (primarily in allocation) of the streams and writers and temporary buffers.

```
private string _path;

[GlobalSetup]
public void Setup() => _path = Path.GetRandomFileName();

[GlobalCleanup]
public void Cleanup() => File.Delete(_path);

[Benchmark]
public void WriteAllText() => File.WriteAllText(_path, Sonnet);
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
WriteAllText	.NET 6.0	488.5 us	1.00	9944 B	1.00
WriteAllText	.NET 7.0	482.9 us	0.99	392 B	0.04

dotnet/runtime#61519 similarly updates `File.ReadAllBytes{Async}` to use `SafeFileHandle` (and `RandomAccess`) directly rather than going through `FileStream`, shaving off some allocation from each use. It also makes the same `SequentialScan` change as mentioned earlier. While this case *is* about reading (whereas the previous change saw `SequentialScan` being complete overhead with no benefit), `ReadAllBytes{Async}` is very frequently used to read smaller files where the overhead of the additional syscall can measure up to 10% of the total cost (and for larger files, modern kernels are pretty good about caching even without a sequentiality hint, so there's little downside measured there).

Another such change is dotnet/runtime#68662, which improved `Path.Join`'s handling of null or empty path segments. `Path.Join` has overloads that accept `string`s and overloads that accept `ReadOnlySpan<char>`s, but all of the overloads produce `string`s. The `string`-based overloads just wrapped each string in a span and delegated to the span-based overloads. However, in the event that the join operation is a nop (e.g. there are two path segments and the second is empty so the join should just return the first), the span-based implementation still needs to create a new string (there's no way for the `ReadOnlySpan<char>`-based overloads to extract a string from the span). As such, the `string`-based overloads can do a little bit better in the case of one of them being null or empty; they can do the same thing the `Path.Combine` overloads do, which is to have the M argument overload delegate to the M-1 argument overload, filtering out a null or empty, and in the base case of the overload with two arguments, if a segment is null or empty, the other (or empty) can just be returned directly.

Beyond that, there are a multitude of allocation-focused PRs, such as dotnet/runtime#69335 from [@pedrobsaila](https://github.com/pedrobsaila) which adds a fast-path based on stack allocation to the internal `ReadLink` helper that's used on Unix anywhere we need to follow symlinks, or dotnet/runtime#68752 that updates `NamedPipeClientStream.ConnectAsync` to remove a delegate allocation (by passing state into a `Task.Factory.StartNew` call explicitly), or dotnet/runtime#69412 which adds an optimized `Read(Span<byte>)` override to the `Stream` returned from `Assembly.GetManifestResourceStream`.

But my personal favorite improvement in this area come from dotnet/runtime#69272, which adds a few new helpers to `Stream`:

```
public void ReadExactly(byte[] buffer, int offset, int count);
public void ReadExactly(Span<byte> buffer);

public ValueTask ReadExactlyAsync(byte[] buffer, int offset, int count, CancellationToken
cancellationToken = default);
public ValueTask ReadExactlyAsync(Memory<byte> buffer, CancellationToken cancellationToken
= default);

public int ReadAtLeast(Span<byte> buffer, int minimumBytes, bool throwOnEndOfStream =
true);
public ValueTask<int> ReadAtLeastAsync(Memory<byte> buffer, int minimumBytes, bool
throwOnEndOfStream = true, CancellationToken cancellationToken = default);
```

In fairness, these are more about usability than they are about performance, but in this case there's a tight correlation between the two. It's very common to write these helpers one's self (the aforementioned PR deleted many open-coded loops for this functionality from across the core libraries) as the functionality is greatly needed, and it's unfortunately easy to get them wrong in ways that negatively impact performance, such as by using a `Stream.ReadAsync` overload that needs to allocate a returned `Task<int>` or reading fewer bytes than is allowed as part of a read call. These implementations are correct and efficient.

Compression

.NET Core 2.1 added support for the Brotli compression algorithm, surfacing it in two ways: `BrotliStream` and the pair of `BrotliEncoder`/`BrotliDecoder` structs that `BrotliStream` is itself built on top of. For the most part, these types just provide wrappers around a native C implementation from google/brotli, and so while the .NET layer has the opportunity to improve how data is moved around, managed allocation, and so on, the speed and quality of the compression itself are largely at the mercy of the C implementation and the intricacies of the Brotli algorithm.

As with many compression algorithms, Brotli provides a knob that allows for a quintessential tradeoff to be made between compression speed (how fast data can be compressed) and compression quality/ratio (how small can the compressed output be made). The hand-wavy idea is the more time the algorithm spends looking for opportunity, the more space can be saved. Many algorithms expose this as a numerical dial, in Brotli's case going from 0 (fastest speed, least compression) to 11 (spend as much time as is needed to minimize the output size). But while `BrotliEncoder` surfaces that same range, `BrotliStream`'s surface area is simpler: most use just specifies that compression should be performed (e.g. `new BrotliStream(destination, CompressionMode.Compress)`) and the only knob available is via the `CompressionLevel` enum (e.g. `new BrotliStream(destination, CompressionLevel.Fastest)`), which provides just a few options: `CompressionLevel.NoCompression`, `CompressionLevel.Fastest`, `CompressionLevel.Optimal`, and `CompressionLevel.SmallestSize`. This means the `BrotliStream` implementation needs to select a default value when no `CompressionLevel` is specified and needs to map `CompressionLevel` to an underlying numerical value when one is.

For better or worse (and I'm about to argue "much worse"), the native C implementation itself defines the default to be 11 (google/brotli#encode.h), and so that's what `BrotliStream` has ended up using when no `CompressionLevel` is explicitly specified. Further, the `CompressionLevel.Optimal` enum value is poorly named. It's intended to represent a good default that's a balanced tradeoff between speed and quality; that's exactly what it means for `DeflateStream`, `GZipStream`, and `ZLibStream`. But for `BrotliStream`, as the default it similarly got translated to mean the underlying native library's default, which is 11. This means that when constructing a `BrotliStream` with either `CompressionMode.Compress` or `CompressionLevel.Optimal`, rather than getting a nice balanced default, you're getting the dial turned all the way up to 11.

Is that so bad? Maybe compression quality is the most important thing? For example, reducing the size of data can make it faster to then transmit it over a wire, and with a slow connection, size then meaningfully translates into end-to-end throughput.

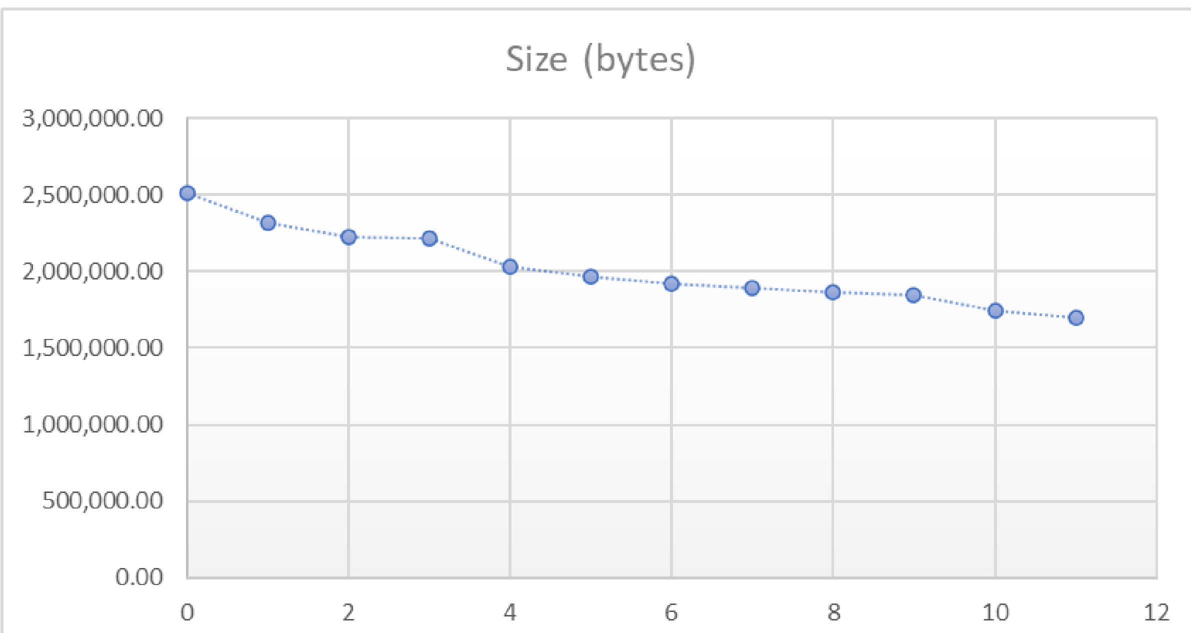
The problem is just how much this extra effort costs. Compression speed and ratio are highly dependent on the data being compressed, so take this example with a small grain of salt as it's not entirely representative of all use, but it's good enough for our purposes. Consider this code, which uses `BrotliEncoder` to compress the [The Complete Works of William Shakespeare from Project Gutenberg]() at varying levels of compression:

```
using System.Buffers;
using System.Diagnostics;
using System.IO.Compression;
using System.Text;

using var hc = new HttpClient();
byte[] data = await hc.GetByteArrayAsync("https://www.gutenberg.org/ebooks/100.txt.utf-8");
Console.WriteLine(data.Length);

var compressed = new MemoryStream();
var sw = new Stopwatch();

for (int level = 0; level <= 11; level++)
{
    const int Trials = 10;

    compressed.Position = 0;
    Compress(level, data, compressed);

    sw.Restart();
    for (int i = 0; i < Trials; i++)
    {
        compressed.Position = 0;
        Compress(level, data, compressed);
    }
    sw.Stop();

    Console.WriteLine($"{level},{sw.Elapsed.TotalMilliseconds /
Trials},{compressed.Position}");

    static void Compress(int level, byte[] data, Stream destination)
    {
        var encoder = new BrotliEncoder(quality: level, window: 22);
        Write(ref encoder, data, destination, false);
        Write(ref encoder, Array.Empty<byte>(), destination, true);
        encoder.Dispose();

        static void Write(ref BrotliEncoder encoder, byte[] data, Stream destination, bool
isFinalBlock)
        {
            byte[] output = ArrayPool<byte>.Shared.Rent(4096);

            OperationStatus lastResult = OperationStatus.DestinationTooSmall;
            ReadOnlySpan<byte> buffer = data;
            while (lastResult == OperationStatus.DestinationTooSmall)
            {
                lastResult = encoder.Compress(buffer, output, out int bytesConsumed, out
int bytesWritten, isFinalBlock);
                if (lastResult == OperationStatus.InvalidData) throw new
InvalidOperationException();
                if (bytesWritten > 0) destination.Write(output.AsSpan(0, bytesWritten));
                if (bytesConsumed > 0) buffer = buffer.Slice(bytesConsumed);
            }

            ArrayPool<byte>.Shared.Return(output);
        }
```

```
    }
}
```

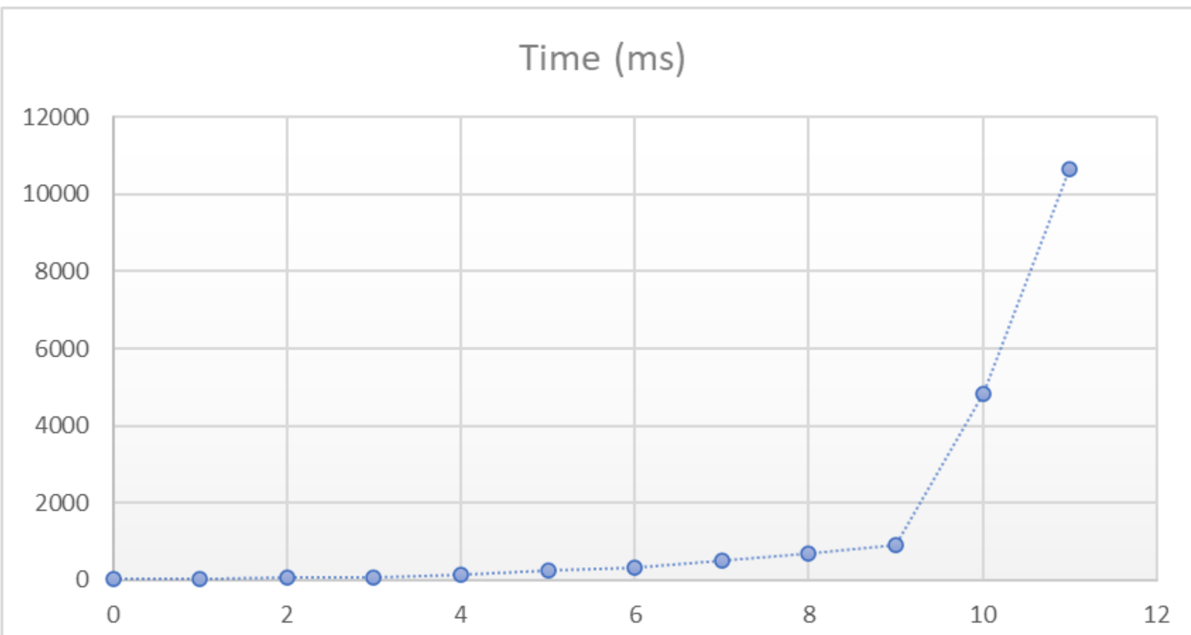
The code is measuring how long it takes to compress the input data at each of the levels (doing a warmup and then averaging several iterations), timing how long it takes and capturing the resulting compressed data size. For the size, I get values like this:

Level	Size (bytes)
0	2,512,855.00
1	2,315,466.00
2	2,224,638.00
3	2,218,328.00
4	2,027,153.00
5	1,964,810.00
6	1,923,456.00
7	1,889,927.00
8	1,863,988.00
9	1,846,685.00
10	1,741,561.00
11	1,702,214.00



That's a fairly liner progression from least to most compression. That's not the problem. This is the problem:

Level	Time (ms)
0	24.11
1	36.67
2	64.13
3	73.72
4	146.41
5	257.12
6	328.54
7	492.81
8	702.38
9	892.08
10	4,830.32
11	10,634.88



This chart shows an almost exponential increase in processing time as we near the upper end of the dial, with quality level 11 compressing ~33% better than quality level 0 but taking ~440x as long to achieve that. If that's what a developer wants, they can specify `CompressionLevel.SmallestSize`, but that cost by default and for the balanced `CompressionLevel.Optimal` is far out of whack.

dotnet/runtime#72266 fixes that. A very small change, it simply makes `CompressMode.Compress` and `CompressionLevel.Optimal` for Brotli map to quality level 4, which across many kinds of inputs does represent a fairly balanced trade-off between size and speed.

```csharp
private byte[] _data = new
HttpClient().GetByteArrayAsync("https://www.gutenberg.org/ebooks/100.txt.utf-8").Result;
private Stream _output = new MemoryStream();

[Benchmark]
public void Compress()
{
    _output.Position = 0;
    using var brotli = new BrotliStream(_output, CompressionMode.Compress, leaveOpen:
true);
    brotli.Write(_data);
}
```

Method	Runtime	Mean	Ratio
Compress	.NET 6.0	9,807.0 ms	1.00
Compress	.NET 7.0	133.1 ms	0.01

Other improvements have gone into compression, such as dotnet/runtime#69439 which updates the internal `ZipHelper.AdvanceToPosition` function used by `ZipArchive` to reuse a buffer on every iteration of a loop rather than allocating a new buffer for each iteration, dotnet/runtime#66764 which uses spans judiciously to avoid a bunch of superfluous `string` and `string[]` allocations from `System.IO.Packaging`, and dotnet/runtime#73082 updating the zlib implementations shipped as part of .NET from v1.2.11 (which was released in January 2017) to v1.2.12 (which was released in March 2022).

CHAPTER 16

Networking

Networking is the life-blood of almost every service, with performance being critical to success. In previous releases, a lot of effort was focused on the lower layers of the networking stack, e.g. .NET 5 saw a significant investment in improving the performance of sockets on Linux. In .NET 7, much of the effort is above sockets.

That said, there were some interesting performance improvements in sockets itself for .NET 7. One of the more interesting is [dotnet/runtime#64770](), which revamped how some synchronization is handled inside of `SocketsAsyncEventArgs`. As background, in the early days of networking in .NET Framework, asynchrony was enabled via `Begin`/`End` methods (the "APM" pattern). This pattern is not only complicated to use well, it's relatively inefficient, resulting in allocation for every single operation performed (at a minimum for the `IAsyncResult` object that's returned from the `BeginXx` method). To help make networking operations more efficient, `SocketsAsyncEventArgs` was introduced. `SocketsAsyncEventArgs` is a reusable class you allocate to hold all of the state associated with asynchronous operations: allocate one, pass it to various async methods (e.g. `ReceiveAsync`), and then completion events are raised on the `SocketAsyncEventArgs` instance when the operation completes. It can be quite efficient when used correctly, but it's also complicated to use correctly. In subsequent releases, `Task`-based and `ValueTask`-based APIs were released; these have the efficiency of `SocketAsyncEventArgs` and the ease-of-use of `async/await`, and are the recommended starting point for all `Socket`-based asynchronous programming today. They have the efficiency of `SocketAsyncEventArgs` because they're actually implemented as a thin veneer on top of it under the covers, and so while most code these days isn't written to use `SocketAsyncEventArgs` directly, it's still very relevant from a performance perspective.

`SocketAsyncEventArgs` on Windows is implemented to use winsock and overlapped I/O. When you call an async method like `ValueTask<Socket> Socket.AcceptAsync(CancellationToken)`, that grabs an internal `SocketAsyncEventArgs` and issues an `AcceptAsync` on it, which in turn gets a `NativeOverlapped*` from the `ThreadPoolBoundHandle` associated with the socket, and uses it to issue the native `AcceptEx` call. When that handle is initially created, we set the `FILE_SKIP_COMPLETION_PORT_ON_SUCCESS` completion notification mode on the socket; use of this was introduced in earlier releases of .NET Core, and it enables a significant number of socket operations, in particular sends and receives, to complete synchronously, which in turn saves unnecessary trips through the thread pool, unnecessary unwinding of async state machines, and so on. But it also causes a condundrum. There are some operations we want to perform associated with asynchronous operation but that have additional overhead, such as registering for the cancellation of those operations, and we don't want to pay the cost of doing them if the operation is going to complete synchronously. That means we really want to delay performing such registration until after we've made the native call and discovered the operation didn't complete synchronously… but at that point we've already initiated the operation, so if it *doesn't* complete synchronously, then we're now in a potential race condition, where our code that's still setting up the asynchronous operation is racing with it potentially completing in a callback on another thread. Fun. `SocketAsyncEventArgs` handled this race condition with a spin lock; the theory was that contention would be incredibly rare, as the vast majority cases would either be the operation completing synchronously (in which case there's no other thread involved) or asynchronously with enough of a delay that the small amount of additional work performed by the initiating thread would have long ago completed by the time the asynchronous operation completed. And for the most part, that was true. However, it turns out that it's actually much more common than expected for certain kinds of operations, like Accepts. Accepts end up almost always completing asynchronously, but if there's already a pending connection, completing asynchronously almost immediately, which then induces this race condition to happen more frequently and results in more contention on the spin locks. Contention on a spin lock is something you really want to avoid. And in fact, for a particular benchmark, this spin lock showed up as the cause for an almost 300% slowdown in requests-per-second (RPS) for a benchmark that used a dedicated connection per request (e.g. with every response setting "Connection: close"). dotnet/runtime#64770 changed the synchronization mechanism to no longer involve a spin lock; instead, it maintains a simple gate implemented as an `Interlocked.CompareExchange`. If the initiating thread gets to the gate first, from that point on the operation is considered asynchronous and any additional work is handled by the completing callback. Conversely, if the callback gets to the gate first, the initiating thread treats the operation as if it completed synchronously. This not only avoids one of the threads spinning while waiting for the other to make forward progress, it also increases the number of operations that end up being handled as synchronous, which in turn reduces other costs (e.g. the code `await`ing the task returned from this operation doesn't need to hook up a callback and exit, and can instead itself continue executing synchronously). The impact of this is difficult to come up with a microbenchmark for, but it can have meaningful impact for loaded Windows servers that end up accepting significant numbers of connections in steady state.

A more-easily quantifiable change around sockets is dotnet/runtime#71090, which improves the performance of `SocketAddress.Equals`. A `SocketAddress` is the serialized form of an `EndPoint`, with a `byte[]` containing the sequence of bytes that represent the address. Its `Equals` method, used to determine whether to `SocketAddress` instances are the same, looped over that `byte[]` byte-by-byte. Not only is such code gratuitous when there are now helpers available like `SequenceEqual` for comparing spans, doing it byte-by-byte is also much less efficient than the vectorized implementation in `SequenceEqual`. Thus, this PR simply replaced the open-coded comparison loop with a call to `SequenceEqual`.

```
private SocketAddress _addr = new IPEndPoint(IPAddress.Parse("123.123.123.123"),
80).Serialize();
private SocketAddress _addr_same = new IPEndPoint(IPAddress.Parse("123.123.123.123"),
80).Serialize();

[Benchmark]
public bool Equals_Same() => _addr.Equals(_addr_same);
```

Method	Runtime	Mean	Ratio
Equals_Same	.NET 6.0	57.659 ns	1.00
Equals_Same	.NET 7.0	4.435 ns	0.08

Let's move up to some more interesting changes in the layers above `Sockets`, starting with `SslStream`.

One of the more impactful changes to `SslStream` on .NET 7 is in support for TLS resumption on Linux. When a TLS connection is established, the client and server engage in a handshake protocol where they collaborate to decide on a TLS version and cipher suites to use, authenticate and validate each other's identity, and create symmetric encryption keys for use after the handshake. This represents a significant portion of the time required to establish a new connection. For a client that might disconnect from a server and then reconnect later, as is fairly common in distributed applications, TLS resumption allows a client and server to essentially pick up where they left off, with the client and/or server storing some amount of information about recent connections and using that information to resume. Windows SChannel provides default support for TLS resumption, and thus the Windows implementation of `SslStream` (which is built on SChannel) has long had support for TLS resumption. But OpenSSL's model requires additional code to enable TLS resumption, and such code wasn't present in the Linux implementation of `SslStream`. With dotnet/runtime#57079 and dotnet/runtime#63030, .NET 7 adds server-side support for TLS resumption (using the variant that *doesn't* require storing recent connection state on the server), and with dotnet/runtime#64369, .NET 7 adds client-side support (which *does* require storing additional state). The effect of this is significant, in particular for a benchmark that opens and closes lots of connections between clients.

```
private NetworkStream _client, _server;
private readonly byte[] _buffer = new byte[1];
private readonly SslServerAuthenticationOptions _options = new
SslServerAuthenticationOptions
{
    ServerCertificateContext = SslStreamCertificateContext.Create(GetCertificate(), null),
};

[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);

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

    _server = new NetworkStream(listener.Accept(), ownsSocket: true);
    _client = new NetworkStream(client, ownsSocket: true);
}

[GlobalCleanup]
public void Cleanup()
{
    _client.Dispose();
    _server.Dispose();
}

[Benchmark]
public async Task Handshake()
{
    using var client = new SslStream(_client, leaveInnerStreamOpen: true, delegate { return
```

```
true; });
    using var server = new SslStream(_server, leaveInnerStreamOpen: true, delegate { return
true; });

    await Task.WhenAll(
        client.AuthenticateAsClientAsync("localhost", null, SslProtocols.None,
checkCertificateRevocation: false),
        server.AuthenticateAsServerAsync(_options));

    await client.WriteAsync(_buffer);
    await server.ReadAsync(_buffer);
    await server.WriteAsync(_buffer);
    await client.ReadAsync(_buffer);
}

private static X509Certificate2 GetCertificate() =>
    new X509Certificate2(

Convert.FromBase64String("MIIUmgIBAzCCFFYGCSqGSIb3DQEHAaCCFEcEghRDMIIUPzCCCiAGCSqGSIb3DQEHA
aCCChEEggoNMIIKCTCCCgUGCyqGSIb3DQEMCgECoIIJfjCCCXowHAYKKoZIhvcNAQwBAzAOBAhCAauyUWggWwICB9AE
gglYefzzX/jx0b+BLU/TkAVj1KBpojf0o6qdTXV42drqIGhX/k1WwF1ypVYdHeeuDfhH2eXHImwPTw+0bACY0dSiIHK
ptm0sb/MskoGI8nlOtHWLi+QBirJ9LSUZcBNOLwoMeYLSFEWWBT69k/sWrc6/SpDoVumkfG4pZ02D9bQgs1+k8fpZjZ
GoZp1jput8CQXPE3JpCsrkdSdiAbWdbNNnYAy4C9Ej/vdyXJVdBTEsKzPYajAzo6Phj/oS/J3hMxxbReMtj2Z0QkoBB
VMc70d+DpAK5OY3et872D5bZjvxhjAYh5JoVTCLTLjbtPRn1g7qh2dQsIpfQ5KrdgqdImshHvxgL92ooC1eQVqQffMn
Z0/LchWNb2rMDa89K9CtAefEIF4ve2bOUZUNFqQ6dvd90SgKq6jNfwQf/1u70WKE86+vChXMMcHFeKso6hTE9+/zuUP
NVmbRefYAtDd7ng996S15FNVdxqyVLlmfcihX1jGhTLi//WuMEaOfXJ9KiwYUyxdUnMp5QJqO8X/tiwnsuhlFe3NKMX
Y77jUe8F7I+dv5cjb9iKXAT+q8oYx1LcWu2mj1ER9/b2omnotp2FIaJDwI40Tts6t4QVH3bUNE9gFIfTMK+WMgKBz/J
AGvC1vbPSdFsWIqwhl7mEYWx83HJp/+Uqp5f+d8m4phSan2rkHEeDjkUaoifLWHWDmL94SZBrgU6yGVK9dU82kr7jCS
UTrnga8qDYsHwpQ22QZtu0aOJGepSwZU7NZNMiyX6QR2hI0CNMjvTK2VusHFB+qnvw+19DzaDT6P0KNPxwBwp07KMQm
3HWTRNt9u6gKUmo5FHngoGte+TZdY66dAwCl0Pt+p1v18XlOB2KOQZKLXnhgikjOwYQxFr3oTb2MjsP6YqnSF9EpYpm
iNySXiYmrYxVinHmK+5JBqoQCN2C3N24slZkYq+AYUTnNST7Ib2We3bBICOFdVUgtFITRW40T+0XZnIv8G1Kbaq/1av
fWI/ieKKxyiYp/ZNXaxc+ycgpsSsAJEuhb83bUkSBpGg9PvFEF0DXm4ah67Ja1SSTmvrCnrOsWZXIpciexMWRGoKrdv
d7Yzj9E8hiu+CGTC4T6+7FxVXJrjCg9zU9G2U6g7uxzoyjGj1wqkhxgvl9pPbz6/KqDRLOHCEwRF4qlWXhsJy4levxG
tifFt6n7DWaNSsOUf8Nwpi+d4fd7LQ7B5tW/y+/vVZziORueruCWO4LnfPhpJ70g18uyN7KyzrWy29rpE46rfjZGGt0
WDZYahObPbw6HjcqSOuzwRoJMxamQb2qsuQnaBS6Bhb5PAnY4SEA045odf/u9uC7mLom2KGNHHz6HrgEPas2UHoJLux
YvY1pza/29akuVQZQUvMA5yMFHHGYZLtTKtCGdVGwX0+QS6ovpV93xux4I/5TrD5U8z9RmTdAx03R3MUhkHF7Zbv5eg
DNsVar+41YWG4VkV1ZXtsZRKJf0hvKNvrpH0e7fVKBdXljm5PXOSg2VdtkhhOpnKKSMcv6MbGWVi/svWLnc7Qim4A4M
Daz+bFVZmh3oGJ7WHvRQhWIcHUL+YJx+064+4IKXZJ/2a/+b2o7C8mJ3GGSBx831ADogg6MRWZx3UY19OZ8YMvpzmZE
BRZZnm4KgNpj+SQnf6pGzD2cmnRhzG60LSNPb17iKbdoUAEMkgt2tlMKXpnt1r7qwsIoTt407cAdCEsUH7OU/AjfFmS
kKJZ7vC5HweqZPnhgJgZ6LYHlfiRzUR1xeDg8JG0nb0vb7LUE4nGPy39/TxIGos7WNwGpG1QVL/8pKjFdjwREaR8e5C
STlQ7gxHV+G3FFvFGpA1p8cRFzlgE6khDLrSJIUkhkHMA3oFwwAzBNIKVXjToyxCogDqxWya0E1Hw5rVCS/zOCS1De2
XQbXs//g46TW0wTJwvgNbs0xLShf3XB+23meeEsMTCR0+igtMMMsh5K/vBUGcJA27ru/KM9qEBcseb/tqCkhhsdj1dn
H0HDmpgFf5DfVrjm+P6ickcF2b+Ojr9t7XHgFszap3COpEPGmeJqNOUTuU53tu/O774IBgqINMWvvG65yQwsEO06jRr
FPRUGb0eH6UM4vC7wbKajnfDuI/EXSgvuOSZ9wE8DeoeK/5We4pN7MSWoDl39gI/LBoNDKFYEYuAw/bhGp8nOwDKki4
a16aYcBGRClpN3ymrdurWsi7TjyFHXfgW8fZe4jXLuKRIk19lmL1gWyD+3bT3mkI2cU2OaY2C0fVHhtiBVaYbxBV8+k
jK8q0Q70zf0r+xMHnewk9APFqUjguPguTdpCoH0VAQST9Mmriv/J12+Y+fL6H+jrtDY2zHPxTF85pA4bBBnLA7Qt9TK
Ce6uuWu5yBqxOV3w2Oa4Pockv1gJzFbVnwlEUWnIjbWVIyo9vo4LBd03uJHPPIQbUp9kCP/Zw+Zblo42/ifyY+a+scw
l1q1dZ7Y0L92yJCKm9Qf6Q+1PBK+uU9pcuVTg/Imqcg5T7jFO5QCi88uwcorgQp+qoeFi0F9tnUecfDl6d0PSgAPnX9
XA0ny3bPwSiWOA8+uW73gesxnGTsNrtc1j85tail8N6m6S2tHXwOmM65J4XRZlzzeM4D/Rzzh13xpRA9kzm9T2cSHsX
EYmSW1X7WovrmYhdOh9K3DPwSyG4tD58cvC7X79UbOB+d17ieo7ZCj+NSLVQO1BqTK0QfErdoVHGKfQG8Lc/ERQRqj1
32Mhi2/r5Ca7AWdqD7/3wgRdQTJSFXt/akpM44xu5DMTCISEFOLWiseSOBtzT6ssaq2Q35dCkXp5wVbWxkXAD7Gm34F
FXXyZrJWAx45Y40wj/0KDJoEzXCuS4Cyiskx1EtYNNOtfDC5wngywmINFUnnW0NkdKSxmDJvrT6HkRKN8ftik7tP4Zv
TaTS28Z0fDmWJ+RjvZW+vtF6mrIzYgGOgdpZwG0ZOSKrXKrY3xpMO16fXyawFfBosLzCty7uA57niPS76UXdbplgPan
IGFyceTg1MsNDsd8vszXd4KezN2VMaxvw+93s0Uk/3Mc+5MAj+UhXPi5UguXMhNo/CU7erzyxYreOlAI7ZzGhPk+oT9
g/MqWa5RpA2IBUaK/wgaNaHChfCcDj/J1qEl6YQQboixxp1IjQxiV9bRQzgwf31Cu2m/FuHTTkPCdxDK156pyFdhcgT
pTNy7RPLDF0MBMGCSqGSIb3DQEJFTEGBAQBAAAAMF0GCSsGAQQBgjcRATFQHk4ATQBpAGMAcgBvAHMAbwBmAHQAIABT
AHQAcgBvAG4AZwAgAEMAcgB5AHAAdABvAGcAcgBhAHAAaABpAGMAIABQAHIAbwB2AGkAZABlAHIwggoXBgkqhkiG9w0
BBwagggoIMIIKBAIBADCCCf0GCSqGSIb3DQEHATAcBgoqhkiG9w0BDAEGMA4ECH63Q8xWHKhqAgIH0ICCCdDAo9x82r
wRM6s16wMo01glVedahn1COCP1FKmP6lQ3kjcHruIWlcKW+eCUpt41qs0LM3iFcPQj5x7675DeLL0AC2Ebu7Jhg0FGM
JZwHLbmJLyG0VSb1WhX2UfxNSdLrdZv8pmejB7DYdV3xAj8DBCRGfwwnbTQjFH9wUPga5U79Dvpqq+YVvUEEci1N6tT
Pu32LOOEvjoEtpskrHoKyqLGV7sSgM6xMIDcfVWbLb8fDcVS1JQRHbeOdGClFMDjwzr+eGWd+OyOZ6BydUGjIKAZpRp
```

```
0YTk5jjYUMNRbvBP1VPq9ASIh8pJnt/Kq1nqfj7EPatXJJUZAH35E6bSbLBnP0+5+xim1l4HsB8066c4B3aTUXnLepP
RyMIn6Xh5ev0pF3aUc4ZlWgar57TzKUFBTkcH5OCbqZloQ7ZCDNc4C3WKVLSUOKLj3QOxJPrb6/nyXZHjki1tGKisb9
RLv4dkeMdRjsSwNRn6Cfdlk2qHWUCiWLlsLXFyMSM12qrSSfIIBRo0wbn1SEJagHqUmlF9UR5A6b5OODIbDq3cXH/q6
U09zVX/BxqxyZqEfeSAcvXjqImLWnZzbIgm0QH7jOtti/vEfvdzypdWH9V64PzQj/5B8P4ZpbQyWUgzKEIdx24WhTOc
dwNivkaEkGFTra3qw2dKO0RTVtx3bSgesHCumQDuDf8yafLfchWuqihYV7zvqW9BWrsa0W7yKNXLNqdlSz8KvuTnFff
OOHrJQwBs+JKdMcKX5IR222RH3fp8Dp17y8hFEaPp4AqpuhHGALXOCwmuPtlUjuHRCUluh3BjaPPLNwLmSGfe0piOVh
4rTyJCfN4rlz0lWBAAfIHi47J9sTnSgEJgkTuemPJXssQ3Z/trcYdfhlYjelOBtS/5DW3wFmjNDilwVBQT66li5xUvc
WvZPx/scXgbgpsMThqguJWtiPLR1SzusKCN4q7bVQ8D8ErHh5uMb5NmNRIZ/xNeqslqTU9A4bi0TE0FjEu28F0Wg4Cx
iwqNM58xik9eni85t+S0Uo9wPV1V2Vdhe9LkO3PeoSTCau4D189DoViL44WPDQ+TCSvlPP7SFEwaBvUlGBWjxJWVb81
lkgRsol1bllUvIzN13V0LSiA0Nks9w9H8cQ17ZRe2r7SpDDR6Rn5oLb9G98AyvlcgJfyUe1iZCUAUZGEU247KwePtXY
AlO47HbAJe0bOtM9zp7KyWxbImKCfxsPWv6CR6PH+ooHDBO9kXVpKaJCYWeYybSMuPufy/u/rMcIVO4oXVsdnjh4jAx
pQOXowCAcN2+Q+XnqtiCr9Mzd0q5ee7jsYuJF6LQRdNP04wIpwjpdggKyB7zURPeTXlV8vIjUs25+CoCxp+fCXfXKqe
2xxdbQ2zFbpKSbJdpbWad3F6MsFBGOKTdyK8EZODGApBtlo71kY6uOxKiBwJKd76zTMsPEQWOZphi2khpTxIVYONrmP
KjSO8zc4dTC8SW+d1kmCt4UYblwoeDCAYp2RiDHpgC+5yuBDCooT/6fG6GQpa1X0PiH2oUCpltZz2M4+1bH2HdTeBfc
1Mtj/hniLL8VdH0qcpS0KYPUxJFEg6IxxrWw1OBreY//6pJLm76nKiflzhz+Mt0RbQZqkPP/K9BxzQw//bW9Kh4iRQ3
7D9HNQG/GtrCEcbH4V4uUjbj34sEo0FC7gVvDob0Bik8l/c901zQZEydqe0DgHtGbY2xIZ2qqsQy4LDVfHNHqSLiNss
L8BJtxUyvnhiwHD7jmyCB6cWyFGtibRBehQzleioS16xvLph88CMGV3IH9By5QtXpDIB4vjhibE6coPkTmpDCB9xlTE
3TV4GBt5JLttkjfOkXAAx0xD523Adcy6FVe5QYuY1O817OO6l88YptozyWi5jVfDh+aDg9pjsw/aZ1hCURe9KDaB4gI
lW4ZEGKsf5e/xU+vuVxw374te/Y2aCChSj93XyC+Fjxe06s4yifVAYA0+HtLMGNHe/X0kPXvRnoa5kIu0yHrzViQrBb
/4Sbms617Gg1BFONks1JO2G0zIt8CouTqVmdtuH7tV0JZV/Nmg7NQ1X59XDC/JH2i4jOu8OhnmIZFlTysS6e1qnqsGt
/0XcUyzPia8+UIAynXmyi8sWlUjy37w6YqapAfcs7B3TezqIwn7RgRasJpNBi7eQQqg5YLe6EYTxctKNkGpzeTBUiXN
XM4Gv3tIaMbzwlhUNbYWuNBsi/7XJPM5jMycINRbdPwYy19gRBs3pm0FoP2Lhl5mVAJ2R8a40Lo5g73wvt9Th+uB9/y
c196RryQe280yfgKiwUoFFcDnL6SoQTRCTl95mF8zw1f3Hc7QImhubgcLntXEndzSNN7ZIDSAB8HiDSR6CGYPNiCNAC
4hj+jUswoWIE257h+deWFTUvjTZmXH+XMoN6trqjdeCH0hePdmrIWVdr1uTIoO16TR6mFNm6Utzc0t5vVrcpnEh3w6a
mVHw5xmweW4S75ncN6vSPxGjtfuQ6c2RTG5NXZuWpnhXwOxgoBN4q/h99zVRvwwsF32Eyzx6GOYLmORgCkzke9eXjjX
WY83oysXx/aE9WCqt3en8zzRzzA1aO9Yi88uv1OOqTvWEoGrf4e7SgjXO6hNjYE6EEvK+mMz6a9F3xSWsUlMsZPIIBe
8CEgNEhXKsa6xw7ljSx8Nz7zYG+u5rgXKFmSNvWvwasZyIfRXkccqODl17BaevbWp/ir3rJ/b9mOiV0UW8qIJ3zC6b1
lXU5pNuOODjqhKkjIHPGXiql+uBPVlfUy8Zbi4AntZAeNIB7HtUavVKX6CF7k9AFtRHIWK70+cFEw4yMZiQjaWeB3dt
16Fz6LZ8+c17kuB2wFuZQqYQkf3quWQVPwKj41gFYoFSwFfJ8L6TBcNHI2u3avtVp9ZbP9zArT8An9Ryri/PwTSbPLT
caz549b60/0k4c/qV4XRMuFsi29CXcMnLSCPpPKs71LTvsRXK6QUJd4fX/KnTiWargbS6tT6lR/bBqY/gFU1xWyKQ8x
ij97vlQjffSKdcbj5JsnjSr8xAh9idfJ2FWZZUJReR9EU1twK7slyUivNLVY7bqroE6CzYaEDecRqfwIrFrzmH+gJoM
88waGRC0JTvm8GpBX0eTb5bnMxJKPtH1GIffgyQLERO1jwjApr6SJEB4yV7x48CZPod9wE51OxUY2hEdAA5l7DBTJys
g5gn/nhY6ZzL0llb39yVyDEcZdmrji0ncEMdBDioGBV3mNz1DL398ZLdjG+xkneI3sgyzgm3cZZ1+/A2kloIEmOKJSe
0k/B1cyMB5QRnXpObF1vWXjauMVIKm0wlLY3YQ9I1vfr6y1o2DN+Vy0sumbIQrjDKqMDswHzAHBgUrDgMCGgQUHEWyD
7i5PbatVl3k0+S9WV3ZJRAEFFd7xcvfj1HpkOawyGnJdtcQ0KWPAgIH0A=="),
        "testcertificate",
        X509KeyStorageFlags.DefaultKeySet);
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Handshake	.NET 6.0	4.647 ms	1.00	19.27 KB	1.00
Handshake	.NET 7.0	2.314 ms	0.50	9.56 KB	0.50

Another significant improvement for `SslStream` in .NET 7 is support for OCSP stapling. When a client handshakes with the server and the server shares its certificate, a client that cares about validating it's talking to exactly who it intended to talk to needs to validate that certificate. In the days of yore, such validation was done with certificate revocation lists (CRL), where periodically the client would download a giant list of certificates known to be revoked. Online Certificate Status Protocol (OCSP) is a newer protocol and mechanism that enables a client to get real-time information about a certificate; while the client handshakes with the server and the server sends the client its certificate, the client then connects to an "OCSP responder" and sends it a request to determine whether the certificate is considered good. OCSP has multiple issues of its own, however. In particular, it places a significant load on these OCSP responder servers, with every client making a real-time request to it about every certificate encountered, and also potentially significantly increasing the time it takes the client to establish a connection. OCSP stapling offers a solution to this. Rather than a client issuing a request to the OCSP responder, the server itself contacts the OCSP responder and gets a signed ticket from the OCSP responder stating that the server's certificate is good and will be for some period of time. When a client handshakes with the server, the server can then "staple" (include) this signed ticket as part of its response to the client, giving the validation to the client directly rather than the client needing to make a separate roundtrip to the OCSP responder. This reduces overheads for everyone involved. dotnet/runtime#67011 adds support for OCSP stapling to `SslStream` client usage on Linux, with dotnet/runtime#69833 adding the Linux server-side counterpart, and dotnet/runtime#71570 adds client-side support for Windows.

The aforementioned changes are primarily about the performance of opening a connection. Additional work has been done to improve that further in other ways. dotnet/runtime#69527 gets rid of allocations associated with several `SafeHandle` instances that were being created unnecessarily on Linux as part of establishing a TLS connection. This highlights the benefits of doing profiling on multiple platforms, as while these `SafeHandle`s were necessary in the Windows implementation, they were fairly meaningless in the Linux implementation (due to differences between SChannel and OpenSSL), and were only brought along for the ride because of how the platform-abstraction layer (PAL) was defined to reuse most of the `SslStream` code across platforms. And dotnet/runtime#68188 avoids several collections allocated as part of the TLS handshake. This one is particularly interesting as it's come up multiple times in the past in various libraries. Imagine you have a lazily initialized property like this:

```
private List<T>? _items;
public List<T> Items => _items ??= new List<T>();
```

And then some code in the same implementation comes along and wants to read the contents of these items. That code might look like:

```
if (Items.Count > 0) { ... }
```

but the very act of accessing `Items` just to check its count forces the collection into existence (with a 0 `Count`). If the code instead checks:

```
if (_items is List<T> items && items.Count > 0) { ... }
```

It can save that unnecessary collection allocation. The approach is made even simpler with C# pattern matching:

```
if (_items is { Count: > 0 }) items) { ... }
```

This is one of those things that's incredibly obvious once you "see" it and realize what's happening, but you often miss until it jumps out at you in a profiler.

dotnet/runtime#69098 is another good example of how profiling can lead to insights about allocations that can be removed. Application-Layer Protocol Negotation (ALPN) allows code establishing a TLS connection to piggy-back on the roundtrips that are being used for the TLS handshake anyway to negotiate some higher-level protocol that will end up being used as well. A very common use-case, for example, is for an HTTPS client/server to negotiate which version of HTTP should be used. This information is exposed from `SslStream` as an `SslApplicationProtocol` struct returned from its `NegotiatedApplicationProtocol` property, but as the actual negotiated protocol can be arbitrary data, `SslApplicationProtocol` just wraps a `byte[]`. The implementation had been dutifully allocating a `byte[]` to hold the bytes passed around as part of ALPN, since we need such a `byte[]` to store in the `SslApplicationProtocol`. But while the byte data *can* be arbitrary, in practice by far the most common byte sequences are equivalent to "http/1.1" for HTTP/1.1, "h2" for HTTP/2, and "h3" for HTTP/3. Thus, it makes sense to special-case those values and use a reusable cached `byte[]` singleton when one of those values is needed. If `SslApplicationProtocol` exposed the underlying `byte[]` directly to consumers, we'd be hesitant to use such singletons, as doing so would mean that if code wrote into the `byte[]` it would potentially be changing the value for other consumers in the same process. However, `SslApplicationProtocol` exposes it as a `ReadOnlyMemory<byte>`, which is only mutable via unsafe code (using the `MemoryMarshal.TryGetArray` method), and once you're employing unsafe code to do "bad" things," all bets are off anyway. [dotnet/runtime#63674](https://github.com/dotnet/runtime/pull/63674) also removes allocations related to ALPN, in this case avoiding the need for a `byte[]` allocation on Linux when setting the negotiated protocol on a client `SslStream`. It uses stack memory instead of an array allocation for protocols up to 256 bytes in length, which is way larger than any in known use, and thus doesn't bother to do anything fancy for the fallback path, which will never be used in practice. And [dotnet/runtime#69103](https://github.com/dotnet/runtime/pull/69103) further avoids ALPN-related allocations and work on Windows by entirely skipping some unnecessary code paths: various methods can be invoked multiple times during a TLS handshake, but even though the ALPN-related work only needed to happen once the first time, the code wasn't special-casing it and was instead repeating the work over and over.

Everything discussed thus far was about establishing connections. What about the performance of reading and writing on that connection? Improvements have been made there, too, in particular around memory management and asynchrony. But first we need some context.

When `async/await` were first introduced, `Task` and `Task<TResult>` were the only game in town; while the pattern-based mechanism the compiler supports for arbitrary "task-like" types enabled `async` methods to return other types, in practice it was only tasks (which also followed our guidance). We soon realized, however, that a significant number of calls to a significant number of commonly-used async APIs would actually complete synchronously. Consider, for example, a method like `MemoryStream.ReadAsync`: `MemoryStream` is backed entirely by an in-memory buffer, so even though the operation is "async," every call to it completes synchronously, as the operation can be performed without doing any potentially long-running I/O. Or consider `FileStream.ReadAsync`. By default `FileStream` employs its own internal buffer. If you issue a call to `FileStream.ReadAsync` with your own buffer and ask for only, say, 16 bytes, under the covers `FileStream.ReadAsync` will issue the actual native call with its own much larger buffer, which by default is 4K. The first time you issue your 16-byte read, actual I/O will be required and the operation is likely to complete asynchronously. But the next 255 calls you make could simply end up draining the remainder of the data read into that 4K buffer, in which case 255 of the 256 "async" operations actually complete synchronously. If the method returns a `Task<int>`, every one of those 255 synchronously-completing calls could still end up allocating a `Task<int>`, just to hand back the `int` that's already known. Various techniques were devised to minimize this, e.g. if the `int` is one of a few well-known values (e.g. -1 through 8), then the async method infrastructure will hand back a pre-allocated and cached `Task<int>` instance for that value, and various stream implementations (including `FileStream`) would cache the previously-returned `Task<int>` and hand it back for the next call as well if the next call yielded exactly the same number of bytes. But those optimizations don't fully mitigate the issue. Instead, we introduced the `ValueTask<TResult>` struct and provided the necessary "builder" to allow `async` methods to return them. `ValueTask<TResult>` was simply a discrimated union between a `TResult` and `Task<TResult>`. If an async method completed asynchronously (or if it failed synchronously), well, it would simply allocate the `Task<TResult>` as it otherwise would have and return that task wrapped in a `ValueTask<TResult>`. But if the method actually completed synchronously and successfully, it would create a `ValueTask<TResult>` that just wrapped the resulting `TResult`, which then eliminates all allocation overhead for the synchronously-completing case. Yay, everyone's happy. Well, almost everyone. For really hot paths, especially those lower down in the stack that many other code paths build on top of, it can also be beneficial to avoid the allocations even for the asynchronously completing case. To address that, .NET Core 2.1 saw the introduction of the `IValueTaskSource<TResult>` interface along with enabling `ValueTask<TResult>` to wrap an instance of that interface in addition to a `TResult` or a `Task<TResult>` (at which point it also became meaningful to introduce a non-generic `ValueTask` and the associated `IValueTaskSource`). Someone can implement this interface with whatever behaviors they want, although we codified the typical implementation of the core async logic into the `ManualResetValueTaskSourceCore` helper struct, which is typically embedded into some object, with the interface methods delegating to corresponding helpers on the struct. Why would someone want to do this? Most commonly, it's to be able to reuse the same instance implementing this interface over and over and over. So, for example, `Socket` exposes a `ValueTask<int> ReceiveAsync` method, and it caches a single instance of an `IValueTaskSource<int>` implementation for use with such receives. As long as you only ever have one receive pending on a given socket at a time (which is the 99.999% case), every `ReceiveAsync` call will either return a `ValueTask<int>` wrapped around an `int` value or a `ValueTask<int>` wrapped around that reusable `IValueTaskSource<int>`, making all use of `ReceiveAsync` ammortized allocation-free (there is another instance used for `SendAsync`, such that you can have a concurrent read and write on the socket and still avoid allocations). However, implementing this support is still non-trivial, and can be super hard when dealing with an operation that's composed of multiple suboperations, which is exactly where `async/await` shine. Thus, C# 10 added support for overriding the default builder that's used on an individual async method (e.g. such that someone could provide their own builder for a `ValueTask<int>`-returning method instead of the one that allocates `Task<int>` instances for asynchronous completion) and .NET 6 included the new `PoolingAsyncValueTaskMethodBuilder` and `PoolingAsyncValueTaskMethodBuilder<>` types. With those, an `async` method like:

```
public async ValueTask<int> ReadAsync(Memory<byte> buffer) { ... }
```

can be changed to be:

```
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))]
public async ValueTask<int> ReadAsync(Memory<byte> buffer) { ... }
```

which will cause the C# compiler to emit the implementation of this method using `PoolingAsyncValueTaskMethodBuilder<int>` instead of the default `AsyncValueTaskMethodBuilder<int>`. The implementation of `PoolingAsyncValueTaskMethodBuilder<TResult>` is true to its name; it employs pooling to avoid *most* of the allocation asynchronous completion would otherwise experience (I say "most" because the pooling by design tries to balance all the various costs involved and may still sometimes allocate), and makes it easy for methods implemented with `async/await` to reap those benefits. So, if this was all introduced in the last release, why am I talking about it now? Pooling isn't free. There are various tradeoffs involved in its usage, and while it can make microbenchmarks look really good, it can also negatively impact real-world usage, e.g. by increasing the cost of garbage collections that do occur by increasing the number of Gen2 to Gen0 references that exist. As such, while the functionality is valuable, we've been methodical in where and how we use it, choosing to do so more slowly and only employing it after sufficient analysis deems it's worthwhile.

Such is the case with `SslStream`. With [dotnet/runtime#69418](), two core and hot `async` methods on `SslStream`'s read path were annotated to use pooling. A microbenchmark shows what I mean when I wrote this can make microbenchmarks look really good (focus on the allocation columns). This benchmark is repeatedly issuing a read (that will be forced to complete asynchronously because there's no available data to satisfy it), then issuing a write to enable that read to complete, and then `await`ing the read's completion; every read thus completes asynchronously.

```
private SslStream _sslClient, _sslServer;
private readonly byte[] _buffer = new byte[1];
private readonly SslServerAuthenticationOptions _options = new
SslServerAuthenticationOptions
{
    ServerCertificateContext = SslStreamCertificateContext.Create(GetCertificate(), null),
};

[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);

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

    _sslClient = new SslStream(new NetworkStream(client, ownsSocket: true),
leaveInnerStreamOpen: true, delegate { return true; });
    _sslServer = new SslStream(new NetworkStream(listener.Accept(), ownsSocket: true),
leaveInnerStreamOpen: true, delegate { return true; });

    Task.WaitAll(
        _sslClient.AuthenticateAsClientAsync("localhost", null, SslProtocols.None,
checkCertificateRevocation: false),
        _sslServer.AuthenticateAsServerAsync(_options));
}

[GlobalCleanup]
public void Cleanup()
{
    _sslClient.Dispose();
    _sslServer.Dispose();
}

[Benchmark]
public async Task ReadWriteAsync()
{
    for (int i = 0; i < 1000; i++)
    {
        ValueTask<int> read = _sslClient.ReadAsync(_buffer);
```

```
        await _sslServer.WriteAsync(_buffer);
        await read;
    }
}
```

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
ReadWriteAsync	.NET 6.0	68.34 ms	1.00	510 B	336404 B	1.000
ReadWriteAsync	.NET 7.0	69.60 ms	1.02	514 B	995 B	0.003

One final change related to reading and writing performance on an `SslStream`. I find this one particularly interesting, as it highlights a new and powerful C# 11 and .NET 7 feature: static abstract members in interfaces. `SslStream`, as with every `Stream`, exposes both synchronous and asynchronous methods for reading and writing. And as you may be aware, the code within `SslStream` for implementing reads and writes is not particularly small. Thus, we really want to avoid having to duplicate all of the code paths, once for synchronous work and once for asynchronous work, when in reality the only place that bifurcation is needed is at the leaves where calls into the underlying `Stream` are made to perform the actual I/O. Historically, we've had two different mechanisms we've employed in [dotnet/runtime](https://github.com/dotnet/runtime) for handling such unification. One is to make all methods `async`, but with an additional `bool useAsync` parameter that gets fed through the call chain, then branching based on it at the leaves, e.g.

```
public static void Work(Stream s) =>
    A(s, useAsyunc: false).GetAwaiter().GetResult(); // GetResult() to propagate any
exceptions

public static Task WorkAsync(Stream S) =>
    A(s, useAsync: true);

internal static async Task A(Stream s, bool useAsync)
{
    ...
    await B(s, useAsync);
    ...
}

private static async Task B(Stream s, bool useAsync)
{
    ...
    int bytesRead = useAsync ?
        await s.ReadAsync(buffer) :
        s.Read(buffer.Span);
    ...
}
```

This way most of the logic and code is shared, and when useAsync is false, everything completes synchronously and so we don't pay for allocation that might otherwise be associated with the `async`-ness. The other approach is similar in spirit, but instead of a `bool` parameter, taking advantage of generic specialization and interface-implementing structs. Consider an interface like:

```
interface IReader
{
    ValueTask<int> ReadAsync(Stream s, Memory<byte> buffer);
}
```

We can then declare two implementations of this interface:

```
struct SyncReader : IReader
{
    public ValueTask<int> ReadAsync(Stream s, Memory<byte> buffer) =>
        new ValueTask<int>(s.Read(buffer.Span));
}

struct AsyncReader : IReader
{
    public ValueTask<int> ReadAsync(Stream s, Memory<byte> buffer) =>
        s.ReadAsync(buffer);
}
```

Then we can redeclare our earlier example as:

```
public static void Work(Stream s) =>
    A(stream, default(SyncReader)).GetAwaiter().GetResult(); // to propagate any exceptions

public static Task WorkAsync(Stream S) =>
    A(s, default(AsyncReader));

internal static async Task A<TReader>(Stream s, TReader reader) where TReader : IReader
{
    ...
    await B(s, reader);
    ...
}

private static async Task B<TReader>(Stream s, TReader reader) where TReader : IReader
{
    ...
    int bytesRead = await reader.ReadAsync(s, buffer);
    ...
}
```

Note that the generic constraint on the `TReader` parameter here allows the implementation to invoke the interface methods, and passing the structs as a generic avoids boxing. One code path supporting both sync and async implementations.

This latter generic approach is how `SslStream` has historically handled the unification of its sync and async implementations. It gets better in .NET 7 with C# 11 now that we have static abstract methods in interfaces. We can instead declare our interface as (note the `static abstract` addition):

```
interface IReader
{
    static abstract ValueTask<int> ReadAsync(Stream s, Memory<byte> buffer);
}
```

our types as (note the `static` addition):

```
struct SyncReader : IReader
{
    public static ValueTask<int> ReadAsync(Stream s, Memory<byte> buffer) =>
        new ValueTask<int>(s.Read(buffer.Span));
}

struct AsyncReader : IReader
```

```
{
    public static ValueTask<int> ReadAsync(Stream s, Memory<byte> buffer) =>
        s.ReadAsync(buffer);
}
```

and our consuming methods as (note the removal of the parameter and the switch to calling static methods on the type parameter):

```
public static void Work(Stream s) =>
    A<SyncReader>(stream).GetAwaiter().GetResult(); // to propagate any exceptions

public static Task WorkAsync(Stream S) =>
    A<AsyncReader>(s);

internal static async Task A<TReader>(Stream s) where TReader : IReader
{
    ...
    await B<TReader>(s);
    ...
}

private static async Task B<TReader>(Stream s) where TReader : IReader
{
    ...
    int bytesRead = await TReader.ReadAsync(s, buffer);
    ...
}
```

Not only is this cleaner, but from a performance perspective we no longer need to pass around the dummy generic parameter, which is general goodness, but for an async method it's particularly beneficial because the state machine type ends up storing all parameters as fields, which means every parameter can increase the amount of allocation incurred by an async method if the method ends up completing asynchronously. dotnet/runtime#65239 flipped `SslStream` (and `NegotiateStream`) to follow this approach. It's also used in multiple other places now throughout dotnet/runtime. dotnet/runtime#69278 from [@teo-tsirpanis](https://github.com/teo-tsirpanis) changed the `RandomAccess` class' implementation for Windows and the `ThreadPool`'s mechanism for invoking work items to use the same approach. Further, dotnet/runtime#63546 did the same in the `Regex` implementation, and in particular in the new `RegexOptions.NonBacktracking` implementation, as a way to abstract over DFA and NFA-based operations using the same code (this technique was since further utilized in `NonBacktracking`, such as by dotnet/runtime#71234 from [@olsaarik](https://github.com/olsaarik)). And potentially most impactfully, dotnet/runtime#73768 did so with `IndexOfAny` to abstract away the differences between `IndexOfAny` and `IndexOfAnyExcept` (also for the `Last` variants). With the introduction of the `{Last}IndexOfAnyExcept` variations previously mentioned, we now have four different variants of `IndexOfAny` with essentially the same functionality: searching forward or backwards, and with equality or inequality. While more challenging to try to unify the directional aspect, this PR utilized this same kind of generic specialization to hide behind an interface the ability to negate the comparison; the core implementations of these methods can then be implemented once and passed either a `Negate` or `DontNegate` implementation of the interface. The net result is not only that the new `Except` varieties immediately gained all of the optimizations of the non-`Except` varieties, but also the goal of trying to make everything consistent resulted in finding places where we were missing optimization opportunities in existing methods (gaps that the PR also rectified).

```
private static readonly string s_haystack = new
HttpClient().GetStringAsync("https://www.gutenberg.org/files/1661/1661-0.txt").Result;

[Benchmark]
public int LastIndexOfAny() => s_haystack.AsSpan().LastIndexOfAny(';', '_');
```

Method	Runtime	Mean	Ratio
LastIndexOfAny	.NET 6.0	9.977 us	1.00
LastIndexOfAny	.NET 7.0	1.172 us	0.12

Let's move up the stack to HTTP. Most of the folks focusing on networking in .NET 7 were focused on taking the preview support for HTTP/3 that shipped in .NET 6 and making it a first-class supported feature in .NET 7. That included functional improvements, reliability and correctness fixes, and performance improvements, such that HTTP/3 can now be used via `HttpClient` on both Windows and Linux (it depends on an underlying QUIC implementation in the msquic component, which isn't currently available for macOS). However, there were significant improvements throughout the HTTP stack, beyond HTTP/3.

One aspect of `HttpClient` that cuts across all versions of HTTP is support for handling and representing headers. While significant improvements went into previous releases to trim down the size of the data structures used to store header information, further work on this front was done for .NET 7. [dotnet/runtime#62981](https://github.com/dotnet/runtime/pull/62981), for example, improves the data structure used to store headers. One of the things `HttpHeaders` needs to deal with is that there's no defined limit to the number of headers that can be sent with an HTTP request or response (though in order to mitigate possible denial of service attacks, the implementation has a configurable limit for how many bytes of headers are accepted from the server), and thus it needs to be able to handle an arbitrary number of them and to do so with efficient access. As such, for the longest time `HttpHeaders` has used a `Dictionary<,>` to provide `O(1)` lookup into these headers. However, while it's valid to have large numbers of headers, it's most common to only have a handful, and for only a few items, the overheads involved in a hash table like `Dictionary<>` can be more than just storing the elements in an array and doing an `O(N)` lookup by doing a linear search through all the elements (algorithmic complexity ignores the "constants" involved, so for a small `N`, an `O(N)` algorithm might be much faster and lighterweight than an `O(1)`). This PR takes advantage of that and teaches `HttpHeaders` how to use either an array or a dictionary; for common numbers of headers (the current threshold is 64), it just uses an array, and in the rare case where that threshold is exceeded, it graduates into a dictionary. This reduces the allocation in `HttpHeader` in all but the most niche cases while also making it faster for lookups.

Another header-related size reduction comes in [dotnet/runtime#64105](https://github.com/dotnet/runtime/pull/64105). The internal representation of headers involves a `HeaderDescriptor` that enables "known headers" (headers defined in the HTTP specifications or that we're otherwise aware of and want to optimize) to share common data, e.g. if a response header matches one of these known headers, we can just use the header name string singleton rather than allocating a new string for that header each time we receive it. This `HeaderDescriptor` accomodated both known headers and custom headers by having two fields, one for known header data (which would be null for custom headers) and one for the header name. Instead, this PR employs a relatively-common technique of having a single `object` field that then stores either the known header information or the name, since the known header information itself includes the name, and thus we don't need the duplication. At the expense of a type check when we need to look up information from that field, we cut the number of fields in half. And while this `HeaderDescriptor` is itself a struct, it's stored in header collections, and thus by cutting the size of the `HeaderDescriptor` in half, we can significantly reduce the size of those collections, especially when many custom headers are involved.

```
private readonly string[] _strings = new[] { "Access-Control-Allow-Credentials", "Access-Control-Allow-Origin", "Cache-Control", "Connection", "Date", "Server" };

[Benchmark]
public HttpResponseHeaders GetHeaders()
{
    var headers = new HttpResponseMessage().Headers;
    foreach (string s in _strings)
    {
        headers.TryAddWithoutValidation(s, s);
    }
    return headers;
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
GetHeaders	.NET 6.0	334.4 ns	1.00	664 B	1.00
GetHeaders	.NET 7.0	213.9 ns	0.64	360 B	0.54

Similarly focused on allocation, [dotnet/runtime#63057](https://github.com/dotnet/runtime/pull/63057) removes two fields from the `HttpHeaderValueCollection<T>` collection type, which provides the concrete implementation for `ICollection<T>` properties like `HttpContentHeaders.ContentEncoding`, `HttpRequestHeaders.UserAgent`, and `HttpResponseHeaders.Server`. The initial design and implementation of this type were overly flexible, with a mechanism for custom validation of values, which entailed multiple fields for storing things like an `Action<>` callback to use for validation. But as it turns out in practice, that validation was only used for one specific consumer, and so rather than making everyone pay for the extra space that wasn't typically used, the validation was instead extracted out to just the call sites it was required.

A more focused allocation reduction comes in [dotnet/runtime#63641](https://github.com/dotnet/runtime/pull/63641). The shared internal utility method `HttpRuleParser.GetHostLength` was using `string.Substring` in order to hand back the parsed host information, but only some of the callers needed this. Rather than making everyone pay for something that not everyone needed, this logic was moved into only the call sites that needed it.

Other small allocation improvements were also made outside of headers. For example, when new HTTP/1 and HTTP/2 connections are created, the implementation queues a work item to the thread pool to handle the actual creation, primarily to escape locks that might be held higher in the call stack. To do so, it used `Task.Run`. And while normally `Task.Run` is a fine thing to use, in this case there were two issues: the resulting `Task` was being ignored, such that any unexpected exceptions would just be eaten, and the lambda being passed to `Task.Run` was closing over `this` and a local, which means the C# compiler will have generated code to allocate both a "display class" (an object to store the state being passed in) for the closure and then also a delegate to a method on that display class. Instead, [dotnet/runtime#68750](https://github.com/dotnet/runtime/pull/68750) switches it to use `ThreadPool.QueueUserWorkItem`, using the overload that takes a generic `TState`, and passing in a tuple of all required state in order to avoid both superfluous allocations.

Folks using HTTP often need to go through a proxy server, and in .NET the ability to go through an HTTP proxy is represented via the `IWebProxy` interface; it has three members, `GetProxy` for getting the `Uri` of the proxy to use for a given destination `Uri`, the `IsBypassed` method which says whether a given `Uri` should go through a proxy or not, and then a `Credentials` property to be used when accessing the target proxy. The canonical implementation of `IWebProxy` provided in the core libraries is the aptly named `WebProxy`. `WebProxy` is fairly simple: you give it a proxy `Uri`, and then calls to `GetProxy` return that proxy `Uri` if the destination isn't to be bypassed. Whether a `Uri` should be bypassed is determined by two things (assuming a non-null proxy `Uri` was provided): did the constructor of the `WebProxy` specify that "local" destinations should be bypassed (and if so, is this destination local), or does this destination address match any of any number of regular expressions provided. As it turns out, this latter aspect has been relatively slow and allocation-heavy in all previous releases of .NET, for two reasons: every call to check whether an address was bypassed was recreating a `Regex` instance for every supplied regular expression, and every call to check whether an address was bypassed was deriving a new `string` from the `Uri` to use to match against the `Regex`. In .NET 7, both of those issues have been fixed, yielding significant improvements if you rely on this regular expression functionality. [dotnet/runtime#73803](https://github.com/dotnet/runtime/pull/73803) from [@onehourlate](https://github.com/onehourlate) changed the handling of the collection of these `Regex` instances. The problem was that `WebProxy` exposes an `ArrayList` (this type goes back to the beginning of .NET and was created pre-generics), which the consumer could modify, and so `WebProxy` had to assume the collection was modified between uses and addressed that by simply creating new `Regex` instances on every use; not good. Instead, this PR creates a custom `ArrayList`-derived type that can track all relevant mutations, and then only if the collection is changed (which is incredibly rare, bordering on never) do the `Regex` instances need to be recreated. And [dotnet/runtime#73807](https://github.com/dotnet/runtime/pull/73807) takes advantage of stack allocation and the `MemoryExtensions.TryWrite` method with string interpolation to format the text into stack memory, avoiding the string allocation. This, combined with the new `Regex.IsMatch(ReadOnlySpan<char>)` overload that enables us to match against that `stackalloc`'d span, makes that aspect of the operation allocation-free as well. Altogether, drastic improvements:

```
private WebProxy _proxy = new WebProxy("http://doesntexist", BypassOnLocal: false, new[] {
@"\.microsoft.com", @"\.dot.net", @"\.bing.com" });
private Uri _destination = new
Uri("https://docs.microsoft.com/dotnet/api/system.net.webproxy");

[Benchmark]
public bool IsBypassed() => _proxy.IsBypassed(_destination);
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
IsBypassed	.NET 6.0	5,343.2 ns	1.00	7528 B	1.00
IsBypassed	.NET 7.0	205.5 ns	0.04	-	0.00

Also related to HTTP, `WebUtility`'s `HtmlDecode` method has improved for .NET 7. The implementation had been manually iterating through each character in the input looking for a `'&'` to be unescaped. Any time you see such an open-coded loop looking for one or more specific characters, it's a red flag that `IndexOf` should be strongly considered. [dotnet/runtime#70700](https://github.com/dotnet/runtime/pull/70700) deletes the entire searching function and replaces it with `IndexOf`, yielding simpler and much faster code (you can see other improvements to use `IndexOf` variants in networking, such as [dotnet/runtime#71137](https://github.com/dotnet/runtime/pull/71137), which used

`IndexOfAny` in `HttpListener`'s `HandleAuthentication` to search a header for certain kinds of whitespace):

```
private string _encoded = WebUtility.HtmlEncode("""
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.
    Condimentum vitae sapien pellentesque habitant. Vitae auctor eu augue ut lectus. Augue
lacus viverra vitae congue eu.
    Tempus quam pellentesque nec nam aliquam sem. Urna nec tincidunt praesent semper
feugiat nibh sed. Amet tellus cras adipiscing
    enim eu. Duis ultricies lacus sed turpis tincidunt. Et sollicitudin ac orci phasellus
egestas tellus rutrum tellus pellentesque.
    """);

[Benchmark]
public string HtmlDecode() => WebUtility.HtmlDecode(_encoded);
```

Method	Runtime	Mean	Ratio
HtmlDecode	.NET 6.0	245.54 ns	1.00
HtmlDecode	.NET 7.0	19.66 ns	0.08

There have been a myriad of other performance-related improvements in networking as well, such as [dotnet/runtime#67881](#) which removed the use of `TcpClient` from `FtpWebRequest`; [dotnet/runtime#68745](#) in `WebSocket` which removed a parameter from one of the core async methods (and since parameters end up on the state machine, if the async method yields this results in fewer allocated bytes); and [dotnet/runtime#70866](#) and [dotnet/runtime#70900](#), which replaced all remaining use of `Marshal.PtrToStructure` in the core networking code with more efficient marshaling (e.g. just performing casts). While `Marshal.PtrToStructure` is valuable when custom marshaling directives are used and the runtime needs to be involved in the conversion, it's also much more heavyweight than just casting, which can be done when the native and managed layouts are bit-for-bit compatible. As with the `u8` example earlier, this comparison is hardly fair, but that's exactly the point:

```
private IntPtr _mem;

[GlobalSetup]
public void Setup()
{
    _mem = Marshal.AllocHGlobal(8);
    Marshal.StructureToPtr(new SimpleType { Value1 = 42, Value2 = 84 }, _mem, false);
}

[GlobalCleanup]
public void Cleanup() => Marshal.FreeHGlobal(_mem);

public struct SimpleType
{
    public int Value1;
    public int Value2;
}

[Benchmark(Baseline = true)]
public SimpleType PtrToStructure() => Marshal.PtrToStructure<SimpleType>(_mem);

[Benchmark]
public unsafe SimpleType Cast() => *(SimpleType*)_mem;
```

Method	Mean	Ratio
PtrToStructure	26.6593 ns	1.000
Cast	0.0736 ns	0.003

For folks using `NegotiateStream`, [dotnet/runtime#71280](https://github.com/dotnet/runtime/pull/71280) from [@filipnavara](https://github.com/filipnavara) will also be very welcome (this comes as part of a larger effort, primarily in [dotnet/runtime#71777](https://github.com/dotnet/runtime/pull/71777) from [@filipnavara](https://github.com/filipnavara) and [dotnet/runtime#70720](https://github.com/dotnet/runtime/pull/70720) from [@filipnavara](https://github.com/filipnavara), to expose the new `NegotiateAuthentication` class). It removes a significant amount of allocation from a typical NTLM handshake by reusing a buffer rather than reallocating a new buffer for each of multiple phases of the handshake:

```
private NetworkStream _client, _server;

[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);

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

    Socket server = listener.Accept();

    _client = new NetworkStream(client, ownsSocket: true);
    _server = new NetworkStream(server, ownsSocket: true);
}

[Benchmark]
public async Task Handshake()
{
    using NegotiateStream client = new NegotiateStream(_client, leaveInnerStreamOpen:
true);
    using NegotiateStream server = new NegotiateStream(_server, leaveInnerStreamOpen:
true);
    await Task.WhenAll(client.AuthenticateAsClientAsync(),
server.AuthenticateAsServerAsync());
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
Handshake	.NET 6.0	1.905 ms	1.00	240.5 KB	1.00
Handshake	.NET 7.0	1.913 ms	1.00	99.28 KB	0.41

CHAPTER 17

JSON

`System.Text.Json` was introduced in .NET Core 3.0, and has seen a significant amount of investment in each release since. .NET 7 is no exception. New features in .NET 7 include support for customizing contracts, polymorphic serialization, support for required members, support for DateOnly / TimeOnly, support for IAsyncEnumerable<T> and JsonDocument in source generation, and support for configuring MaxDepth in JsonWriterOptions. However, there have also been new features specifically focused on performance, and other changes about improving performance of JSON handling in a variety of scenarios.

One of the biggest performance pitfalls we've seen developers face with `System.Text.Json` has to do with how the library caches data. In order to achieve good serialization and deserialization performance when the source generator isn't used, `System.Text.Json` uses reflection emit to generate custom code for reading/writing members of the types being processed. Instead of then having to pay reflection invoke costs on every access, the library incurs a much larger one-time cost per type to perform this code generation, but then all subsequent handling of these types is very fast… assuming the generated code is available for use. These generated handlers need to be stored somewhere, and the location that's used for storing them is them is `JsonSerializerOptions`. The idea was intended to be that developers would instantiate an options instance once and pass it around to all of their serialization/deserialization calls; thus, state like these generated handlers could be cached on them. And that works well when developers follow the recommended model. But when they don't, performance falls off a cliff, and hard. Instead of "just" paying for the reflection invoke costs, each use of a new `JsonSerializerOptions` ends up re-generating via reflection emit those handlers, skyrocketing the cost of serialization and deserialization. A super simple benchmark makes this obvious:

```
private JsonSerializerOptions _options = new JsonSerializerOptions();
private MyAmazingClass _instance = new MyAmazingClass();

[Benchmark(Baseline = true)]
public string ImplicitOptions() => JsonSerializer.Serialize(_instance);

[Benchmark]
public string WithCached() => JsonSerializer.Serialize(_instance, _options);

[Benchmark]
public string WithoutCached() => JsonSerializer.Serialize(_instance, new
JsonSerializerOptions());

public class MyAmazingClass
{
    public int Value { get; set; }
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
ImplicitOptions	.NET 6.0	170.3 ns	1.00	200 B	1.00
WithCached	.NET 6.0	163.8 ns	0.96	200 B	1.00
WithoutCached	.NET 6.0	100,440.6 ns	592.48	7393 B	36.97

In .NET 7, this was fixed in [dotnet/runtime#64646](https://github.com/dotnet/runtime/pull/64646) (and subsequently tweaked in [dotnet/runtime#66248](https://github.com/dotnet/runtime/pull/66248)) by adding a global cache of the type information separate from the options instances. A `JsonSerializerOptions` still has a cache, but when new handlers are generated via reflection emit, those are also cached at the global level (with appropriate removal when no longer used in order to avoid unbounded leaks).

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
ImplicitOptions	.NET 6.0	170.3 ns	1.00	200 B	1.00
ImplicitOptions	.NET 7.0	166.8 ns	0.98	48 B	0.24
WithCached	.NET 6.0	163.8 ns	0.96	200 B	1.00
WithCached	.NET 7.0	168.3 ns	0.99	48 B	0.24
WithoutCached	.NET 6.0	100,440.6 ns	592.48	7393 B	36.97
WithoutCached	.NET 7.0	590.1 ns	3.47	337 B	1.69

As can be seen here, it's still more expensive to create a new `JsonSerializerOptions` instance on each call, and the recommended approach is "don't do that." But if someone does do it, in this example they're only paying 3.6x the cost rather than 621x the cost, a huge improvement. [dotnet/runtime#61434](https://github.com/dotnet/runtime/pull/61434) also now exposes the `JsonSerializerOptions.Default` instance that's used by default if no options are explicitly provided.

Another change to `JsonSerializer` came in [dotnet/runtime#72510](https://github.com/dotnet/runtime/pull/72510), which slightly improved the performance of serialization when using the source generator. The source generator emits helpers for performing the serialization/deserialization work, and these are then invoked by `JsonSerializer` via delegates (as part of abstracting away all the different implementation strategies for how to get and set members on the types being serialized and deserialized). Previously, these helpers were being emitted as static methods, which in turn meant that the delegates were being created to static methods. Delegates to instance methods are a bit faster to invoke than delegates to static methods, so this PR made a simple few-line change for the source generator to emit these as instance methods instead.

Yet another for `JsonSerializer` comes in [dotnet/runtime#73338](https://github.com/dotnet/runtime/pull/73338), which improves allocation with how it utilizes `Utf8JsonWriter`. `Utf8JsonWriter` is a class, and every time `JsonSerializer` would write out JSON, it would allocate a new `Utf8JsonWriter` instance. In turn, `Utf8JsonWriter` needs something to write to, and although the serializer was using an `IBufferWriter` implementation that pooled the underlying `byte[]` instances employed, the implementation of `IBufferWriter` itself is a class that `JsonSerializer` would allocate. A typical `Serialize` call would then end up allocating a few extra objects and an extra couple of hundred bytes just for these helper data structures. To address that, this PR takes advantage of `[ThreadStatic]`, which can be put onto static fields to make them per-thread rather than per-process. From whatever thread is performing the (synchronous)

`Serialize` operation, it then ensures the current thread has a `Utf8JsonWriter` and `IBufferWriter` instance it can use, and uses them; for the most part this is straightforward, but it needs to ensure that the serialization operation itself doesn't try to recursively serialize, in which case these objects could end up being used erroneously while already in use. It also needs to make sure that the pooled `IBufferWriter` doesn't hold on to any of its `byte[]`s while it's not being used. That instance gets its arrays from `ArrayPool<T>`, and we want those arrays to be usable in the meantime by anyone else making use of the pool, not sequestered off in this cached `IBufferWriter` implementation. This optimization is also only really meaningful for small object graphs being serialized, and only applies to the synchronous operations (asynchronous operations would require a more complicated pooling mechanism, since the operation isn't tied to a specific thread, and the overhead of such complication would likely outweigh the modest gain this optimization provides).

```
private byte[] _data = new byte[] { 1, 2, 3, 4, 5 };

[Benchmark]
public string SerializeToString() => JsonSerializer.Serialize(_data);
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
SerializeToString	.NET 6.0	146.4 ns	1.00	200 B	1.00
SerializeToString	.NET 7.0	137.5 ns	0.94	48 B	0.24

`Utf8JsonWriter` and `Utf8JsonReader` also saw several improvements directly. [dotnet/runtime#69580](https://github.com/dotnet/runtime/pull/69580) adds a few new performance-focused members, the `ValueIsEscaped` property (which exposes already tracked information and enables consumers to avoid the expense of re-checking) and the `CopyString` method (which provides a non-allocating mechanism to get access to a string value from the reader). It then also uses the added support internally to speed up certain operations on `Utf8JsonReader`. And [dotnet/runtime#63863](https://github.com/dotnet/runtime/pull/63863), [dotnet/runtime#71534](https://github.com/dotnet/runtime/pull/71534), and [dotnet/runtime#61746](https://github.com/dotnet/runtime/pull/61746) fix how some exception checks and throws were being handled so as to not slow down the non-exceptional fast paths.

XML

System.Xml is used by a huge number of applications and services, but ever since JSON hit the scene and has been all the rage, XML has taken a back seat and thus hasn't seen a lot of investment from either a functionality or performance perspective. Thankfully, System.Xml gets a bit of performance love in .NET 7, in particular around reducing allocation on some commonly used code paths.

Sometimes a performance fix is as easy as changing a single number. That's the case with [dotnet/runtime#63459](https://github.com/dotnet/runtime/pull/63459) from [@chrisdcmoore](https://github.com/chrisdcmoore), which addresses a long-standing issue with the asynchronous methods on the popular `XmlReader`. When `XmlReader` was originally written, whoever developed it chose a fairly common buffer size to be used for read operations, namely 4K or 8K `char`s depending on various conditions. When `XmlReader` later gained asynchronous reading functionality, for whatever reason a much, much larger buffer size of 64K `char`s was selected (presumably in hopes of minimizing the number of asynchronous operations that would need to be employed, but the actual rationale is lost to history). A key problem with such a buffer size, beyond it leading to a lot of allocation, is the allocation it produces typically ends up on the Large Object Heap (LOH). By default, under the expectation that really large objects are long-lived, objects greater than 85K bytes are allocated into the LOH, which is treated as part of Gen 2, and that makes such allocation if *not* long-lived even more expensive in terms of overall impact on the system. Well, 64K `char`s is 128K bytes, which puts it squarely above that threshold. This PR lowers the size from 64K `char`s to 32K `char`s, putting it below the threshold (and generally reducing allocation pressure, how much memory needs to be zero'd, etc). While it's still a very large allocation, and in the future we could look at pooling the buffer or employing a smaller one (e.g. no different from what's done for the synchronous APIs), this simple one-number change alone makes a substantial difference for shorter input documents (while not perceivably negatively impacting larger ones).

```
private readonly XmlReaderSettings _settings = new XmlReaderSettings { Async = true };
private MemoryStream _stream;

[Params(10, 1_000_000)]
public int ItemCount;

[GlobalSetup]
public void Setup()
{
    _stream = new MemoryStream();
    using XmlWriter writer = XmlWriter.Create(_stream);
    writer.WriteStartElement("Items");
    for (var i = 0; i < ItemCount; i++)
    {
        writer.WriteStartElement($"Item{i}");
        writer.WriteEndElement();
    }
    writer.WriteEndElement();
```

```
}

[Benchmark]
public async Task XmlReader_ReadAsync()
{
    _stream.Position = 0;
    using XmlReader reader = XmlReader.Create(_stream, _settings);
    while (await reader.ReadAsync());
}
```

Method	Runtime	ItemCount	Mean	Ratio	Allocated	Alloc Ratio
XmlReader_ReadAsync	.NET 6.0	10	42.344 us	1.00	195.94 KB	1.00
XmlReader_ReadAsync	.NET 7.0	10	9.992 us	0.23	99.94 KB	0.51
XmlReader_ReadAsync	.NET 6.0	1000000	340,382.953 us	1.00	101790.34 KB	1.00
XmlReader_ReadAsync	.NET 7.0	1000000	333,417.347 us	0.98	101804.45 KB	1.00

`XmlReader` and `XmlWriter` saw other allocation-related improvements as well. [dotnet/runtime#60076](https://github.com/dotnet/runtime/pull/60076) from [@kronic](https://github.com/kronic) improved the `ReadOnlyTernaryTree` internal type that's used when `XmlOutputMethod.Html` is specified in the `XmlWriterSettings`. This included using a `ReadOnlySpan<byte>` initialized from an RVA static instead of a large `byte[]` array that would need to be allocated. And [dotnet/runtime#60057](https://github.com/dotnet/runtime/pull/60057) from [@kronic](https://github.com/kronic), which converted ~400 `string` creations in the `System.Private.Xml` assembly to use interpolated strings. Many of these cases were stylistic, converting something like `string1 + ":" + string2` into `$"{string1}:{string2}"`; I say stylistic here because the C# compiler will generate the exact same code for both of those, a call to `string.Concat(string1, ":", string2)`, given that there's a `Concat` overload that accepts three strings. However, some of the changes do impact allocation. For example, the private `XmlTextWriter.GeneratePrefix` method had the code:

```
return "d" + _top.ToString("d", CultureInfo.InvariantCulture)
     + "p" + temp.ToString("d", CultureInfo.InvariantCulture);
```

where `_top` and `temp` are both `int`s. This will result in allocating two temporary strings and then concatenating those with the two constant strings. Instead, the PR changed it to:

```
return string.Create(CultureInfo.InvariantCulture, $"d{_top:d}p{temp:d}");
```

which while shorter is also more efficient, avoiding the intermediate string allocations, as the custom interpolated string handler used by `string.Create` will format those into a pooled buffer rather than allocating intermediate temporaries.

`XmlSerializer` is also quite popular and also gets a (small) allocation reduction, in particular for deserialization. `XmlSerializer` has two modes for generating serialization/deserialization routines: using reflection emit to dynamically generate IL at run-time that are tuned to the specific shape of the types being serialized/deserialized, and the [XML Serializer Generator Tool](https://learn.microsoft.com/dotnet/core/additional-tools/xml-serializer-generator) (sgen), which generates a .dll containing the same support, just ahead-of-time (a sort-of precursor to the Roslyn source generators we love today). In both cases, when deserializing, the generated code wants to track which properties of the object being deserialized have already been set, and to do that, it uses a `bool[]` as a bit array. Every time an object is deserialized, it allocates a `bool[]` with enough elements to track every member of the type. But in common usage, the vast majority of types being deserialized only have a relatively small number of properties, which means we can easily use stack memory to track this information rather than heap memory. That's what dotnet/runtime#66914 does. It updates both of the code generators to `stackalloc` into a `Span<bool>` for less than or equal to 32 values, and otherwise fall back to the old approach of heap-allocating the `bool[]` (which can also then be stored into a `Span<bool>` so that the subsequent code paths simply use a span instead of an array). You can see this quite easily in the .NET Object Allocation Tracking tool in Visual Studio. For this console app (which, as an aside, shows how lovely the new raw string literals feature in C# is for working with XML):

```
using System.Text;
using System.Xml.Serialization;

var serializer = new XmlSerializer(typeof(Release[]));
var stream = new MemoryStream(Encoding.UTF8.GetBytes(
    """
    <?xml version="1.0" encoding="utf-8"?>
    <ArrayOfRelease xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <Release><Major>1</Major><Minor>0</Minor></Release>
        <Release><Major>1</Major><Minor>1</Minor></Release>
        <Release><Major>2</Major><Minor>0</Minor></Release>
        <Release><Major>2</Major><Minor>1</Minor></Release>
        <Release><Major>2</Major><Minor>2</Minor></Release>
        <Release><Major>3</Major><Minor>0</Minor></Release>
        <Release><Major>3</Major><Minor>1</Minor></Release>
        <Release><Major>5</Major><Minor>0</Minor></Release>
        <Release><Major>6</Major><Minor>0</Minor></Release>
        <Release><Major>7</Major><Minor>0</Minor></Release>
    </ArrayOfRelease>
    """));

for (int i = 0; i < 1000; i++)
{
    stream.Position = 0;
    serializer.Deserialize(stream);
}

public class Release
{
    public int Major;
    public int Minor;
    public int Build;
    public int Revision;
}
```

Here's what I see when I run this under .NET 6:

Type	Allocations
System.Xml.NameTable.Entry	33,000
System.String	29,123
System.Boolean[]	10,013
Release	10,000

We're running a thousand deserializations, each of which will deserialize 10 `Release` instances, and so we expect to see 10,000 `Release` objects being allocated, which we do… but we also see 10,000 `bool[]` being allocated. Now with .NET 7 (note the distinct lack of the per-object `bool[]`):

Allocations Call Tree Functions Collections

Type	Allocations
System.Xml.NameTable.Entry	33,000
System.String	28,964
Release	10,000

Other allocation reduction went into the creation of the serializer/deserializer itself, such as with [dotnet/runtime#68738](https://github.com/dotnet/runtime/pull/68738) avoiding allocating strings to escape text that didn't actually need escaping, [dotnet/runtime#66915](https://github.com/dotnet/runtime/pull/66915) using stack allocation for building up small text instead of using a `StringBuilder`, [dotnet/runtime#66797](https://github.com/dotnet/runtime/pull/66797) avoiding delegate and closure allocations in accessing the cache of serializers previously created, [dotnet/runtime#67001](https://github.com/dotnet/runtime/pull/67001) from [@TrayanZapryanov](https://github.com/TrayanZapryanov) caching an array used with `string.Split`, and [dotnet/runtime#67002](https://github.com/dotnet/runtime/pull/67002) from [@TrayanZapryanov](https://github.com/TrayanZapryanov) that changed some parsing code to avoid a `string.ToCharArray` invocation.

For folks using XML schema, [dotnet/runtime#66908](https://github.com/dotnet/runtime/pull/66908) replaces some `Hashtable`s in the implementation where those collections were storing `int`s as the value. Given that `Hashtable` is a non-generic collection, every one of those `int`s was getting boxed, resulting in unnecessary allocation overhead; these were fixed by replacing these `Hashtable`s with `Dictionary<..., int>` instances. (As an aside, this is a fairly common performance-focused replacement to do, but you need to be careful as `Hashtable` has a few behavioral differences from `Dictionary<,>`; beyond the obvious difference of `Hashtable` returning `null` from its indexer when a key isn't found and `Dictionary<,>` throwing in that same condition, `Hashtable` is thread-safe for use with not only multiple readers but multiple readers concurrent with a single writer, and `Dictionary<,>` is not.) [dotnet/runtime#67045](https://github.com/dotnet/runtime/pull/67045) reduces allocation of `XmlQualifiedName` instances in the implementation of `XsdBuilder.ProcessElement` and `XsdBuilder.ProcessAttribute`. And [dotnet/runtime#64868](https://github.com/dotnet/runtime/pull/64868) from [@TrayanZapryanov](https://github.com/TrayanZapryanov) uses stack-based memory and pooling to avoid temporary string allocation in the implementation of the internal `XsdDateTime` and `XsdDuration` types, which are used by the public `XmlConvert`.

```
private TimeSpan _ts = TimeSpan.FromMilliseconds(12345);

[Benchmark]
public string XmlConvertToString() => XmlConvert.ToString(_ts);
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
XmlConvertToString	.NET 6.0	90.70 ns	1.00	184 B	1.00
XmlConvertToString	.NET 7.0	59.21 ns	0.65	40 B	0.22

XML pops up in other areas as well, as in the `XmlWriterTraceListener` type. While the `System.Diagnostics.Trace` type isn't the recommended tracing mechanism for new code, it's widely used in existing applications, and `XmlWriterTraceListener` let's you plug in to that mechanism to write out XML logs for traced information. dotnet/runtime#66762 avoids a bunch of string allocation occurring as part of this tracing, by formatting much of the header information into a span and then writing that out rather than `ToString()`'ing each individual piece of data.

```
[GlobalSetup]
public void Setup()
{
    Trace.Listeners.Clear();
    Trace.Listeners.Add(new XmlWriterTraceListener(Stream.Null));
}

[Benchmark]
public void TraceWrite()
{
    Trace.WriteLine("Something important");
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
TraceWrite	.NET 6.0	961.9 ns	1.00	288 B	1.00
TraceWrite	.NET 7.0	772.2 ns	0.80	64 B	0.22

CHAPTER 19

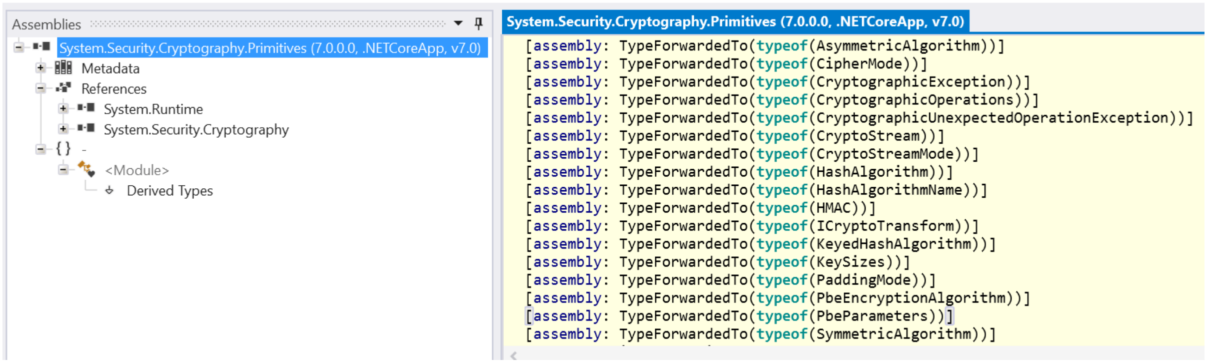
Cryptography

Some fairly significant new features came to `System.Security.Cryptography` in .NET 7, including the support necessary to enable the previously discussed OCSP stapling and support for building certificate revocation lists, but there was also a fair amount of effort put into making existing support faster and more lightweight.

One fairly substantial change in .NET 7 is split across dotnet/runtime#61025, dotnet/runtime#61137, and dotnet/runtime#64307. These PRs don't change any code materially, but instead consolidate all of the various cryptography-related assemblies in the core libraries into a single `System.Security.Cryptography` assembly. When .NET Core was first envisioned, a goal was to make it extremely modular, and large swaths of code were teased apart to create many smaller assemblies. For example, cryptographic functionality was split between `System.Security.Cryptography.Algorithms.dll`, `System.Security.Cryptography.Cng.dll`, `System.Security.Cryptography.Csp.dll`, `System.Security.Cryptography.Encoding.dll`, `System.Security.Cryptography.OpenSsl.dll`, `System.Security.Cryptography.Primitives.dll`, and `System.Security.Cryptography.X509Certificates.dll`. You can see this if you look in your shared framework folder for a previous release, e.g. here's mine for .NET 6:

These PRs move all of that code into a single `System.Security.Cryptography.dll` assembly. This has several benefits. First, crypto is used in a huge number of applications, and most apps would end up requiring multiple (or even most) of these assemblies. Every assembly that's loaded adds overhead. Second, a variety of helper files had to be compiled into each assembly, leading to overall larger amount of compiled code to be distributed. And third, we weren't able to implement everything as optimal as we'd have otherwise liked due to functionality in one assembly not exposed to another (and we avoid using `InternalsVisibleTo` as it hampers maintainability and impedes other analysis and optimizations). Now in .NET 7, the shared framework looks more like this:

Interesting, you still see a bunch of assemblies there, but all except for `System.Security.Cryptography.dll` are tiny; that's because these are simple facades. Because we need to support binaries built for .NET 6 and earlier running on .NET 7, we need to be able to handle binaries that refer to types in these assemblies, but in .NET 7, those types actually live in `System.Security.Cryptography.dll`. .NET provides a solution for this in the form of the `[TypeForwardedTo(...)]` attribute, which enables one assembly to say "hey, if you're looking for type X, it now lives over there." And if you crack open one of these assemblies in a tool like ILSpy, you can see they're essentially empty except for a bunch of these attributes:



In addition to the startup and maintenance wins this provides, this has also enabled further subsequent optimization. For example, there's a lot of object cloning that goes on in the innards of this library. Various objects are used to wrap native handles to OS cryptographic resources, and to handle lifetime semantics and ownership appropriately, there are many cases where a native handle is duplicated and then wrapped in one or more new managed objects. In some cases, however, the original resource is then destroyed because it's no longer needed, and the whole operation could have been made more efficient if the original resource just had its ownership transferred to the new objects rather than being duplicated and destroyed. This kind of ownership transfer typically is hard to do between assemblies as it generally requires public API that's not focused on such usage patterns, but with internals access, this can be overcome. dotnet/runtime#72120 does this, for example, to reduce allocation of various resources inside the `RSACng`, `DSACng`, `ECDsaCng`, and `ECDiffieHellmanCng` public types.

In terms of actual code improvements, there are many. One category of improvements is around "one-shot" operations. With many forms of data processing, all of the data needn't be processed in one operation. A block of data can be processed, then another, then another, until finally there's no more data to be processed. In such usage, there's often some kind of state carried over from the processing of one block to the processing of the next, and then the processing of the last block is special as it needn't carry over anything and instead needs to perform whatever work is required to end the whole operation, e.g. outputting any final footer or checksum that might be required as part of the format. Thus, APIs that are able to handle arbitrary number of blocks of data are often a bit more expensive in one way, shape, or form than APIs that only support a single input; this latter category is known as "one shot" operations, because they do everything in "one shot." In some cases, one-shot operations can be significantly cheaper, and in other cases they merely avoid some allocations that would have been necessary to transfer state from the processing of one block of data to the next. dotnet/runtime#58270 from [@vcsjones](https://github.com/vcsjones) and dotnet/runtime#65725 from [@vcsjones](https://github.com/vcsjones) both improved the performance of various one-shot operations on "symmetric" cryptograhic algorithms (algorithms that use the same key information to both encrypt and decrypt), like AES. The former does so by refactoring the implementations to avoid some reset work that's not necessary in the case of one-shots because the relevant state is about to go away, anyway, and that in turns also allows the implementation to store less of certain kinds of state. The latter does so for decryption one-shots by decrypting directly into the destination buffer whenever possible, using stack space if possible when going directly into the user's buffer isn't feasible, etc.

```
private byte[] _plaintext = Encoding.UTF8.GetBytes("This is a test. This is only a test.
Nothing to see here.");
private byte[] _iv = Enumerable.Range(0, 16).Select(i => (byte)i).ToArray();
private Aes _aes = Aes.Create();
private byte[] _output = new byte[1000];

[Benchmark]
public bool OneShot() => _aes.TryEncryptCfb(_plaintext, _iv, _output, out _);
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
OneShot	.NET 6.0	1.828 us	1.00	336 B	1.00
OneShot	.NET 7.0	1.770 us	0.97	184 B	0.55

In addition to making one-shots lighterweight, other PRs have then used these one-shot operations in more places in order to simplify their code and benefit from the increased performance, e.g. dotnet/runtime#70639 from [@vcsjones](https://github.com/vcsjones), dotnet/runtime#70857 from [@vcsjones](https://github.com/vcsjones), dotnet/runtime#64005 from [@vcsjones](https://github.com/vcsjones), and dotnet/runtime#64174 from [@vcsjones](https://github.com/vcsjones).

There's also a large number of PRs that have focused on removing allocations from around the crypto stack:

- **Stack allocation**. As has been seen in many other PRs referenced throughout this post, using `stackalloc` is a very effective way to get rid of array allocations in many situations. It's used effectively in multiple crypto PRs to avoid either temporary or pooled array allocations, such as in dotnet/runtime#64584 from [@vcsjones](https://github.com/vcsjones), dotnet/runtime#69831 from [@vcsjones](https://github.com/vcsjones), dotnet/runtime#70173 from [@vcsjones](https://github.com/vcsjones), dotnet/runtime#69812 from [@vcsjones](https://github.com/vcsjones), and dotnet/runtime#69448 from [@vcsjones](https://github.com/vcsjones). Sometimes this is used when calling an API that has multiple overloads, including one taking an array and one taking a span. Othertimes it's used with P/Invokes that often just pass out a small amount of data. Sometimes it's used to avoid temporary array allocations, and sometimes it's used in places where pooling was used previously, but the data is often small enough to avoid even the overheads of pooling.
- **Avoiding double copies**. Most of the crypto APIs that accept `byte[]`s and store them end up making defensive copies of those arrays rather than storing the original. This is fairly common throughout .NET, but it's especially common in the crypto stack, where the ability to trust the data is as you expect it (and validate it) is paramount. In some cases, though, code ends up allocating a temporary `byte[]` just to pass data into one of these APIs that copies and re-allocates, anyway. dotnet/runtime#71102 from [@vcsjones](https://github.com/vcsjones), dotnet/runtime#69024 from [@vcsjones](https://github.com/vcsjones), dotnet/runtime#71015 from [@vcsjones](https://github.com/vcsjones), and dotnet/runtime#69534 from [@vcsjones](https://github.com/vcsjones) deal with that duplication in some cases by extracting a span to the original data instead of creating a temporary `byte[]`; when that span is passed into the target API, the target API still makes a copy, but we've avoided the first one and thus cut the array allocation for these operations effectively in half. dotnet/runtime#71888 from [@vcsjones](https://github.com/vcsjones) is a variation on this theme, improving the internals of

`Rfc2898DeriveBytes` to supports spans such that its constructors that accept spans can then do the more efficient thing.

- **Replacing `O(1)` data structures**. `O(1)` lookup data structures like `Dictionary<,>` and `HashSet<>` are the lifeblood of most applications and services, but sometimes algorithmic complexity is misleading. Yes, these provide very efficient searching, but there's still overhead associated with computing a hash code, mapping that hash code to a location in the data structure, and so on. If there's only ever a handful of items (i.e. the `N` in the complexity is really, really small), it can be much faster to just do a linear search, and if `N` is sufficiently small, a data structure may not even be needed at all: the search can just be open-coded as a waterfall of if/elseif/else constructs. That's the case in a PR like [dotnet/runtime#71341](https://github.com/dotnet/runtime/pull/71341) from [@vcsjones](https://github.com/vcsjones), where the 99.999% case involves just five strings (names of hash algorithms); it's cheaper to just compare against each than it is do a `HashSet<>.Contains`, especially since the JIT now unrolls and vectorizes the comparison against the constant string names.
- **Simply avoiding unnecessary work**. The best optimizations are ones where you simply stop doing work you don't have to do. [dotnet/runtime#68553](https://github.com/dotnet/runtime/pull/68553) from [@vcsjones](https://github.com/vcsjones) is a good example of this. This code was performing a hash of some data in order to determine the length of resulting hashes for that particular configuration, but we actually know ahead of time exactly how long a hash for a given algorithm is going to be, and we already have in this code a cascading if/elseif/else that's checking for each known algorithm, so we can instead just hardcode the length for each. [dotnet/runtime#70589](https://github.com/dotnet/runtime/pull/70589) from [@vcsjones](https://github.com/vcsjones) is another good example, in the same spirit of the ownership transfer example mentioned earlier (but this one didn't previously span assembly boundaries). Rather than in several places taking an `X509Extension`, serializing it to a `byte[]`, and passing that temporary `byte[]` to something else that in turn makes a defensive copy, we can instead provide an internal pathway for ownership transfer, bypassing all of the middle stages. Another good one is [dotnet/runtime#70618](https://github.com/dotnet/runtime/pull/70618) from [@vcsjones](https://github.com/vcsjones), as it's an example of how it pays to really understand your dependencies. The implementation of symmetric encryption on macOS uses the CommonCrypto library. One of the functions it exposes is `CCCryptorFinal`, which is used at the end of the encryption/decryption process. However, there are several cases called out in the docs where it's unnecessary ("superfluous," according to the docs), and so our dutifully calling it even in those situations is wasteful. The fix? Stop doing unnecessary work.
- **New APIs**. A bunch of new APIs were introduced for cryptography in .NET 7. Most are focused on easing scenarios that were difficult to do correctly before, like [dotnet/runtime#66509](https://github.com/dotnet/runtime/pull/66509) from [@vcsjones](https://github.com/vcsjones) that provides an `X500DistinguishedNameBuilder`. But some are focused squarely on performance. [dotnet/runtime#57835](https://github.com/dotnet/runtime/pull/57835) from [@vcsjones](https://github.com/vcsjones), for example, exposes a new `RawDataMemory` property on `X509Certificate2`. Whereas the existing `RawData` property returns a new `byte[]` on every call (again a defensive copy to avoid having to deal with the possiblity that the consumer mucked with the raw data), this new `RawDataMemory` returns a `ReadOnlyMemory<byte>` around the internal `byte[]`. Since the only way to access and mutate that underlying `byte[]` via a `ReadOnlyMemory<byte>` is via unsafe interop code (namely via the `System.Runtime.InteropServices.MemoryMarshal` type), it doesn't create a defensive copy and enables accessing this data freely without additional allocation.

Diagnostics

Let's turn our attention to System.Diagnostics, which encompasses types ranging from process management to tracing.

The `Process` class is used for a variety of purposes, including querying information about running processes, interacting with other processes (e.g. being notified of their exiting), and launching processes. The performance of querying for information in particular had some notable improvements in .NET 7. `Process` provides several APIs for querying for process information, one of the most common being `Process.GetProcessesByName`: apps that know the name of the process they're interested in can pass that to `GetProcessesByName` and get back a `Process[]` containing a `Process` for each. It turns out that previous releases of .NET were loading the full information (e.g. all of its threads) about every `Process` on the machine in order to filter down to just those with the target name. [dotnet/runtime#68705](https://github.com/dotnet/runtime/pull/68705) fixes that by only loading the name for a process rather than all of the information for it. While this helps a bit with throughput, it helps a ton with allocation:

```
[Benchmark]
public void GetProcessesByName()
{
    foreach (Process p in Process.GetProcessesByName("dotnet.exe"))
        p.Dispose();
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
GetProcessesByName	.NET 6.0	2.287 ms	1.00	447.86 KB	1.000
GetProcessesByName	.NET 7.0	2.086 ms	0.90	2.14 KB	0.005

Accessing various pieces of information from a `Process` has also improved. If you load a `Process` object via the `Process.GetProcesses` or `Process.GetProcessesByName` methods, by design they load all information about the `Process` being retrieved; internally their state will be populated such that subsequent accesses to members of the `Process` instance will be very fast. But, if you access a `Process` via `Process.GetProcessById` or `Process.GetCurrentProcess` (which is effectively `GetProcessById` for the current process' id), no information other than the process' ID is prepopulated, and the state for the `Process` instance is queried on-demand. In most cases, accessing a single member of one of those lazy-loaded `Process` instances triggers loading all of the data for it, as the information is all available as part of the same native operation, e.g. on Windows using `NtQuerySystemInformation` and on Linux reading from `/proc/pid/stat` and `/proc/pid/status`. But in some cases we can be more fine-grained about it, using APIs that serve up a subset of the data much more quickly. [dotnet/runtime#59672](https://github.com/dotnet/runtime/pull/59672) from [@SteveDunn](https://github.com/SteveDunn) provides one such optimization, using the `QueryFullProcessImageName` on Windows to read the process name in response to `Process.ProcessName` being used. If all you care about reading is the process' name, it's a huge boost in throughput, and even if you subsequently go on to read additional state from the `Process` and force it to load everything else, accessing the process name is so fast that it doesn't add meaningful overhead to the all-up operation. This is visible in this benchmark:

```
[Benchmark]
public string GetCurrentProcessName()
{
    using Process current = Process.GetCurrentProcess();
    return current.ProcessName;
}

[Benchmark]
public string GetCurrentProcessNameAndWorkingSet()
{
    using Process current = Process.GetCurrentProcess();
    return $"{current.ProcessName} {current.WorkingSet64}";
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
GetCurrentProcessName	.NET 6.0	3,070.54 us	1.00	3954 B	1.00
GetCurrentProcessName	.NET 7.0	32.30 us	0.01	456 B	0.12
GetCurrentProcessNameAndWorkingSet	.NET 6.0	3,055.70 us	1.00	4010 B	1.00
GetCurrentProcessNameAndWorkingSet	.NET 7.0	3,149.92 us	1.03	4186 B	1.04

Interestingly, this PR had a small deficiency we didn't initially catch, which is that the `QueryFullProcessImageName` API we switched to didn't work in the case of elevated/privileged processes. To accomodate those, [dotnet/runtime#70073](https://github.com/dotnet/runtime/pull/70073) from [@schuettecarsten](https://github.com/schuettecarsten) updated the code to keep both the new and old implementations, starting with the new one and then only falling back to the old if operating on an incompatible process.

Several additional PRs helped out the `Process` class. When launching processes with `Process.Start` on Unix, the implementation was using `Encoding.UTF8.GetBytes` as part of argument handling, resulting in a temporary array being allocated per argument; dotnet/runtime#71279 removes that per-argument allocation, instead using `Encoding.UTF8.GetByteCount` to determine how large a space is needed and then using the `Encoding.UTF8.GetBytes` overload that accepts a span to encode directly into the native memory already being allocated. dotnet/runtime#71136 simplifies and streamlines the code involved in getting the "short name" of a process on Windows for use in comparing process names. And dotnet/runtime#45690 replaces a custom cache with use of `ArrayPool` in the Windows implementation of getting all process information, enabling effective reuse of the array that ends up being used rather than having it sequestered off in the `Process` implementation forever.

Another area of performance investment has been in `DiagnosticSource`, and in particular around enumerating through data from `Activity` instances. This work translates into faster integration and interoperability via OpenTelemetry, in order to be able to export data from .NET `Activity` information faster. dotnet/runtime#67012 from [@CodeBlanch](https://github.com/CodeBlanch), for example, improved the performance of the internal `DiagLinkedList<T>.DiagEnumerator` type that's the enumerator returned when enumerating `Activity.Links` and `Activity.Events` by avoiding a copy of each `T` value:

```
private readonly Activity _activity;

public Program()
{
    using ActivitySource activitySource = new ActivitySource("Perf7Source");
    ActivitySource.AddActivityListener(new ActivityListener
    {
        ShouldListenTo = s => s == activitySource,
        Sample = (ref ActivityCreationOptions<ActivityContext> o) =>
ActivitySamplingResult.AllDataAndRecorded
    });

    _activity = activitySource.StartActivity(
        "TestActivity",
        ActivityKind.Internal,
        parentContext: default,
        links: Enumerable.Range(0, 1024).Select(_ => new ActivityLink(default)).ToArray());
    _activity.Stop();
}

[Benchmark(Baseline = true)]
public ActivityLink EnumerateActivityLinks()
{
    ActivityLink last = default;
    foreach (ActivityLink link in _activity.Links) last = link;
    return last;
}
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
EnumerateActivityLinks	.NET 6.0	19.62 us	1.00	64 B	1.00
EnumerateActivityLinks	.NET 7.0	13.72 us	0.70	32 B	0.50

Then [dotnet/runtime#67920](https://github.com/dotnet/runtime/pull/67920) from [@CodeBlanch](https://github.com/CodeBlanch) and [dotnet/runtime#68933](https://github.com/dotnet/runtime/pull/68933) from [@CodeBlanch](https://github.com/CodeBlanch) added new `EnumerateTagObjects`, `EnumerateEvents`, and `EnumerateLinks` enumeration methods that return a struct-based enumerator that has a `ref T`-returning `Current` to avoid yet another layer of copy.

```
private readonly Activity _activity;

public Program()
{
    using ActivitySource activitySource = new ActivitySource("Perf7Source");
    ActivitySource.AddActivityListener(new ActivityListener
    {
        ShouldListenTo = s => s == activitySource,
        Sample = (ref ActivityCreationOptions<ActivityContext> o) =>
ActivitySamplingResult.AllDataAndRecorded
    });

    _activity = activitySource.StartActivity(
        "TestActivity",
        ActivityKind.Internal,
        parentContext: default,
        links: Enumerable.Range(0, 1024).Select(_ => new ActivityLink(default)).ToArray());
```

```
    _activity.Stop();
}

[Benchmark(Baseline = true)]
public ActivityLink EnumerateActivityLinks_Old()
{
    ActivityLink last = default;
    foreach (ActivityLink link in _activity.Links) last = link;
    return last;
}

[Benchmark]
public ActivityLink EnumerateActivityLinks_New()
{
    ActivityLink last = default;
    foreach (ActivityLink link in _activity.EnumerateLinks()) last = link;
    return last;
}
```

Method	Mean	Ratio	Allocated	Alloc Ratio
EnumerateActivityLinks_Old	13.655 us	1.00	32 B	1.00
EnumerateActivityLinks_New	2.380 us	0.17	-	0.00

Of course, when it comes to diagnostics, anyone who's ever done anything with regards to timing and measurements is likely familiar with good ol' `Stopwatch`. `Stopwatch` is a simple type that's very handy for getting precise measurements and is thus used all over the place. But for folks that are really cost-sensitive, the fact that `Stopwatch` is a class can be prohibitive, e.g. writing:

```
Stopwatch sw = Stopwatch.StartNew();
...;
TimeSpan elapsed = sw.Elapsed;
```

is easy, but allocates a new object just to measure. To address this, `Stopwatch` has for years exposed the static `GetTimestamp()` method which avoids that allocation, but consuming and translating the resulting `long` value is complicated, requiring a formula involving using `Stopwatch.Frequency` and `TimeSpan.TicksPerSecond` in the right incantation. To make this pattern easy, dotnet/runtime#66372 adds a static `GetElapsedTime` method that handles that conversion, such that someone who wants that last mile of performance can write:

```
long timestamp = Stopwatch.GetTimestamp();
...
TimeSpan elapsed = Stopwatch.GetElapsedTime(timestamp);
```

which avoids the allocation and saves a few cycles:

```
[Benchmark(Baseline = true)]
public TimeSpan Old()
{
    Stopwatch sw = Stopwatch.StartNew();
    return sw.Elapsed;
}

[Benchmark]
public TimeSpan New()
{
```

```
    long timestamp = Stopwatch.GetTimestamp();
    return Stopwatch.GetElapsedTime(timestamp);
}
```

Method	Mean	Ratio	Allocated	Alloc Ratio
Old	32.90 ns	1.00	40 B	1.00
New	26.30 ns	0.80	-	0.00

Exceptions

It might be odd to see the subject of "exceptions" in a post on performance improvements. After all, exceptions are by their very nature meant to be "exceptional" (in the "rare" sense), and thus wouldn't typically contribute to fast-path performance. Which is a good thing, because fast-paths that throw exceptions in the common case are no longer fast: throwing exceptions is quite expensive.

Instead, one of the things we *do* concern ourselves with is how to minimize the impact of checking for exceptional conditions: the actual exception throwing may be unexpected and slow, but it's super common to need to check for those unexpected conditions, and that checking should be very fast. We also want such checking to minimally impact binary size, especially if we're going to have many such checks all over the place, in generic code for which we end up with many copies due to generic specialization, in functions that might be inlined, and so on. Further, we don't want such checks to impede other optimizations; for example, if I have a small function that wants to do some argument validation and would otherwise be inlineable, I likely don't want the presence of exception throwing to invalidate the possibility of inlining.

Because of all of that, high-performance libraries often come up with custom "throw helpers" they use to achieve their goals. There are a variety of patterns for this. Sometimes a library will just define its own static method that handles constructing and throwing an exception, and then call sites do the condition check and delegate to the method if throwing is needed:

```
if (arg is null)
    ThrowArgumentNullException(nameof(arg));
...
[DoesNotReturn]
private static void ThrowArgumentNullException(string arg) =>
    throw new ArgumentNullException(arg);
```

This keeps the IL associated with the throwing out of the calling function, minimizing the impact of the throw. That's particularly valuable when additional work is needed to construct the exception, e.g.

```
private static void ThrowArgumentNullException(string arg) =>
    throw new ArgumentNullException(arg, SR.SomeResourceMessage);
```

Other times, libraries will encapsulate both the checking and throwing. This is exactly what the ArgumentNullException.ThrowIfNull method that was added in .NET 6 does:

```
public static void ThrowIfNull([NotNull] object? argument,
[CallerArgumentExpression("argument")] string? paramName = null)
{
    if (argument is null)
        Throw(paramName);
}
```

```
[DoesNotReturn]
internal static void Throw(string? paramName) => throw new
ArgumentNullException(paramName);
```

With that, callers benefit from the concise call site:

```
public void M(string arg)
{
    ArgumentNullException.ThrowIfNull(arg);
    ...
}
```

the IL remains concise, and the assembly generated for the JIT will include the streamlined condition check from the inlined `ThrowIfNull` but won't inline the `Throw` helper, resulting in effectively the same code as if you'd written the previously shown manual version with `ThrowArgumentNullException` yourself. Nice.

Whenever we introduce new public APIs in .NET, I'm particularly keen on seeing them used as widely as possible. Doing so serves multiple purposes, including helping to validate that the new API is usable and fully addresses the intended scenarios, and including the rest of the codebase benefiting from whatever that API is meant to provide, whether it be a performance improvement or just a reduction in routinely written code. In the case of `ArgumentNullException.ThrowIfNull`, however, I purposefully put on the brakes. We used it in .NET 6 in several dozen call sites, but primarily just in place of custom `ThrowIfNull`-like helpers that had sprung up in various libraries around the runtime, effectively deduplicating them. What we didn't do, however, was replace the literally thousands of null checks we have with calls to `ArgumentNullException.ThrowIfNull`. Why? Because the new `!!` C# feature was right around the corner, destined for C# 11.

For those unaware, the `!!` feature enabled putting `!!` onto parameter names in member signatures, e.g.

```
public void Process(string name!!)
{
    ...
}
```

The C# compiler then compiled that as equivalent to:

```
public void Process(string name)
{
    ArgumentNullException.ThrowIfNull(name);
}
```

(albeit using its own `ThrowIfNull` helper injected as internal into the assembly). Armed with the new feature, [dotnet/runtime#64720]() and [dotnet/runtime#65108]() rolled out use of `!!` across [dotnet/runtime](), replacing ~25,000 lines of code with ~5000 lines that used `!!`. But, what's the line from Kung Fu Panda, "One often meets his destiny on the road he takes to avoid it"? The presence of that initial PR kicked off an unprecedented debate about the `!!` feature, with many folks liking the concept but a myriad of different opinions about exactly how it should be exposed, and in the end, the only common ground was to cut the feature. In response, [dotnet/runtime#68178]() undid all usage of `!!`, replacing most of it with `ArgumentNullException.ThrowIfNull`. There are now ~5000 uses of `ArgumentNullException.ThrowIfNull` across [dotnet/runtime](), making it one of our most popular

APIs internally. Interestingly, while we expected a peanut-buttery effect of slight perf improvements in many places, our performance auto-analysis system flagged several performance improvements (e.g. dotnet/perf-autofiling-issues#3531) as stemming from these changes, in particular because it enabled the JIT's inlining heuristics to flag more methods for inlining.

With the success of `ArgumentNullException.ThrowIfNull` and along with its significant roll-out in .NET 7, .NET 7 also sees the introduction of several more such throw helpers. dotnet/runtime#61633, for example, adds an overload of `ArgumentNullException.ThrowIfNull` that works with pointers. dotnet/runtime#64357 adds the new `ArgumentException.ThrowIfNullOrEmpty` helper as well as using it in several hundred places. And dotnet/runtime#58684 from [@Bibletoon](https://github.com/Bibletoon) adds the new `ObjectDisposedException.ThrowIf` helper (tweaked by dotnet/runtime#71544 to help ensure it's inlineable), which is then used at over a hundred additional call sites by dotnet/runtime#71546.

Registry

On Windows, the Registry is a database provided by the OS for applications and the system itself to load and store configuration settings. Practically every application accesses the registry. I just tried a simple console app:

```
Console.WriteLine("Hello, world");
```

built it as release, and then ran the resulting .exe. That execution alone triggered 64 `RegQueryValue` operations (as visible via SysInternals' [Process Monitor] tool). The core .NET libraries even access the registry for a variety of purposes, such as for gathering data for `TimeZoneInfo`, gathering data for various calendars like `HijriCalendar` and `JapaneseCalendar`, or for serving up environment variables as part of `Environment.GetEnvironmentVariable(EnvironmentVariableTarget)` with `EnvironmentVariableTarget.User` or `EnvironmentVariableTarget.Machine`.

It's thus beneficial to streamline access to registry data on Windows, in particular for reducing overheads in startup paths where the registry is frequently accessed. dotnet/runtime#66918 does just that. Previously, calling `RegistryKey.GetValue` would make a call to `RegQueryValueEx` with a null buffer; this tells the `RegQueryValueEx` method that the caller wants to know how big a buffer is required in order to store the value for the key. The implementation would then allocate a buffer of the appropriate size and call `RegQueryValueEx` again, and for values that are to be returned as strings, would then allocate a string based on the data in that buffer. This PR instead recognizes that the vast majority of data returned from calls to the registry is relatively small. It starts with a `stackalloc`'d buffer of 512 bytes, and uses that buffer as part of the initial call to `RegQueryValueEx`. If the buffer was sufficiently large, we no longer have to make a second system call to retrieve the actual data: we already got it. If the buffer was too small, we rent an `ArrayPool` buffer of sufficient size and use that pooled buffer for the subsequent `RegQueryValueEx` call. Except in situations where we actually need to return a `byte[]` array to the caller (e.g. the type of the key is `REG_BINARY`), this avoids the need for the allocated `byte[]`. And for keys that return strings (e.g. the type of the key is `REG_SZ`), previously the old implementation would have allocated a temporary `char[]` to use as the buffer passed to `RegQueryValueEx`, but we can instead just reinterpret cast (e.g. `MemoryMarshal.Cast`) the original buffer (whether a `stackalloc`'d span or the rented buffer as a `Span<char>`), and use that to construct the resulting string.

```
private static readonly RegistryKey s_netFramework =
Registry.LocalMachine.OpenSubKey(@"SOFTWARE\Microsoft\.NETFramework");

[Benchmark] public string RegSz() => (string)s_netFramework.GetValue("InstallRoot");
```

Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
RegSz	.NET 6.0	6.266 us	1.00	200 B	1.00
RegSz	.NET 7.0	3.182 us	0.51	96 B	0.48

Analyzers

The ability to easily plug custom code, whether for analyzers or source generators, into the Roslyn compiler is one of my favorite features in all of C#. It means the developers working on C# don't need to be solely responsible for highlighting every possible thing you might want to diagnose in your code. Instead, library authors can write their own analyzers, ship them either in dedicated nuget packages or as side-by-side in nuget packages with APIs, and those analyzers augment the compiler's own analysis to help developers write better code. We ship a large number of analyzer rules in the .NET SDK, many of which are focused on performance, and we augment that set with more and more analyzers every release. We also work to apply more and more of those rules against our own codebases in every release. .NET 7 is no exception.

One of my favorite new analyzers was added in [dotnet/roslyn-analyzers#5594](https://github.com/dotnet/roslyn-analyzers/pull/5594) from [@NewellClark](https://github.com/NewellClark) (and tweaked in [dotnet/roslyn-analyzers#5972](https://github.com/dotnet/roslyn-analyzers/pull/5972)). In my [.NET 6 performance](https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6) post, I talked about some of the overheads possible when types aren't sealed:

- Virtual calls are more expensive than regular non-virtual invocation and generally can't be inlined, since the JIT doesn't know what is the actual type of the instance and thus the actual target of the invocation (at least not without assistance from PGO). But if the JIT can see that a virtual method is being invoked on a sealed type, it can devirtualize the call and potentially even inline it.
- If a type check (e.g. `something is typeof(SomeType)`) is performed where `SomeType` is sealed, that check can be implemented along the lines of `something is not null && something.GetType() == typeof(SomeType)`. In contrast, if `SomeType` is not sealed, the check is going to be more along the lines of `CastHelpers.IsInstanceOfClass(typeof(SomeType), something)`, where `IsInstanceOfClass` is a non-trivial (and today non-inlined) call into a JIT helper method in Corelib that not only checks for null and for direct equality with the specified type, but also linearly walks the parent hierarchy of the type of the object being tested to see if it might derive from the specified type.
- Arrays in .NET are covariant, which means if types `B` and `C` both derive from type `A`, you can have a variable typed as `A[]` that's storing a `B[]`. Since `C` derives from `A`, it's valid to treat a `C` as an `A`, but if the `A[]` is actually a `B[]`, storing a `C` into that array would mean storing a `C` into a `B[]`, which is invalid. Thus, every time you store an object reference into an array of reference types, additional validation may need to be performed to ensure the reference being written is compatible with the concrete type of the array in question. But, if `A` in this example were sealed, nothing could derive from it, so storing objects into it doesn't require such covariance checks.
- Spans shift this covariance check to their constructor; rather than performing the covariance check on every write into the array, the check is performed when a span is being constructed from an array, such that if you try to create a `new Span<A>(bArray)`, the ctor will throw an exception. If `A` is sealed, the JIT is able to elide such a check as well.

It effectively would be impossible for an analyzer to be able to safely recommend sealing public types. After all, it has no knowledge of the type's purpose, how it's intended to be used, and whether anyone outside of the assembly containing the type actually derives from it. But internal and private types are another story. An analyzer *can* actually see every possible type that could be deriving from a private type, since the analyzer has access to the whole compilation unit containing that type, and it needn't worry about compatibility because anything that could derive from such a type necessarily must also be non-public and would be recompiled right along with the base type. Further, with the exception of assemblies annotated as InternalsVisibleTo, an analyzer can have the same insight into internal types. Thus, this PR adds CA1852, an analyzer that flags in non-InternalsVisibleTo assemblies all private and internal types that aren't sealed and that have no types deriving from them and recommends they be sealed. (Due to some current limitations in the infrastructure around fixers and how this analyzer had to be written in order to be able to see all of the types in the assembly, the analyzer for CA1852 doesn't show up in Visual Studio. It can, however, be applied using the `dotnet format` tool. And if you bump up the level of the rule from info to warning or error, it'll show up as part of builds as well.)

In .NET 6, we sealed over 2300 types, but even with that, this analyzer ended up finding more to seal. [dotnet/runtime#59941](#) from [@NewellClark](https://github.com/NewellClark) sealed another ~70 types, and [dotnet/runtime#68268](#) which enabled the rule as an warning in [dotnet/runtime](#) (which builds with warnings-as-errors) sealed another ~100 types. As a larger example of the rule in use, ASP.NET hadn't done much in the way of sealing types in previous releases, but with CA1852 now in the .NET SDK, [dotnet/aspnetcore#41457](#) enabled the analyzer and sealed more than ~1100 types.

Another new analyzer, CA1854, was added in [dotnet/roslyn-analyzers#4851](#) from [@CollinAlpert](https://github.com/CollinAlpert) and then enabled in [dotnet/runtime#70157](#). This analyzer looks for the surprisingly common pattern where a `Dictionary<TKey, TValue>`'s `ContainsKey` is used to determine whether a dictionary contains a particular entry, and then if it does, the dictionary's indexer is used to retrieve the value associated with the key, e.g.

```
if (_dictionary.ContainsKey(key))
{
    var value = _dictionary[key];
    Use(value);
}
```

Dictionary's `TryGetValue` method already combines both of these operations, both looking up the key and retrieving its value if it exists, doing so as a single operation:

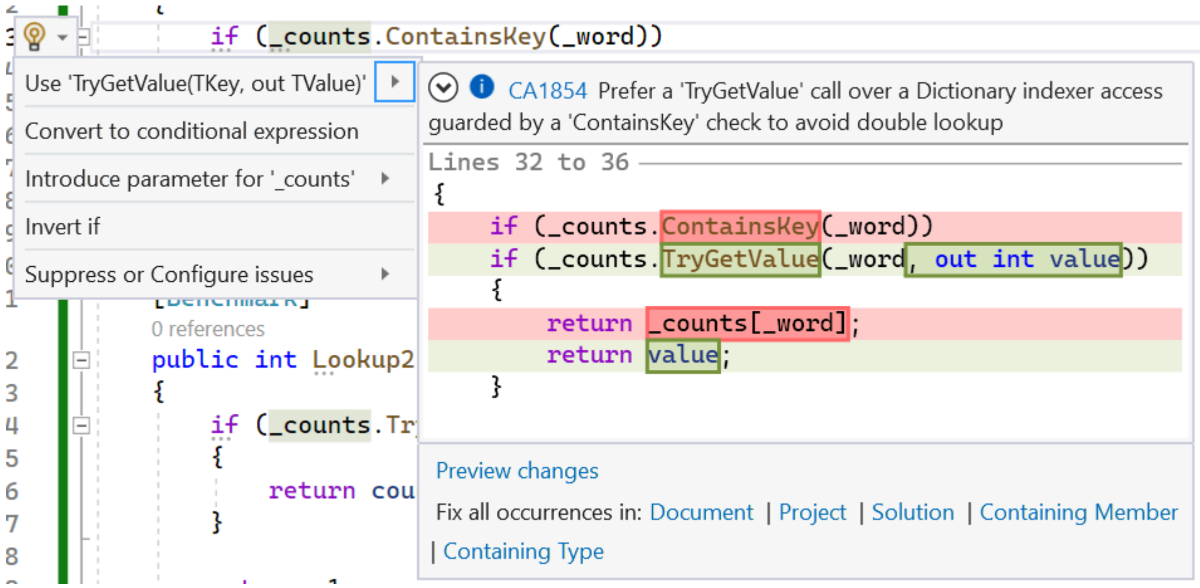
```
if (_dictionary.TryGetValue(key, out var value))
{
     Use(value);
}
```

A benefit of this, in addition to arguably being simpler, is that it's also faster. While `Dictionary<TKey, TValue>` provides very fast lookups, and while the performance of those lookups has gotten faster over time, doing fast work is still more expensive than doing no work, and if we can do one lookup instead of two, that can result in a meaningful performance boost, in particular if it's being performed on a fast path. And we can see from this simple benchmark that looks up a word in a dictionary that, for this operation, making distinct calls to `ContainsKey` and the indexer does indeed double the cost of using the dictionary, almost exactly:

```
private readonly Dictionary<string, int> _counts = Regex.Matches(
    new
HttpClient().GetStringAsync("https://www.gutenberg.org/cache/epub/100/pg100.txt").Result,
@"\b\w+\b")
    .Cast<Match>()
    .GroupBy(word => word.Value, StringComparer.OrdinalIgnoreCase)
    .ToDictionary(word => word.Key, word => word.Count(),
StringComparer.OrdinalIgnoreCase);

private string _word = "the";

[Benchmark(Baseline = true)]
public int Lookup1()
{
    if (_counts.ContainsKey(_word))
    {
        return _counts[_word];
    }

    return -1;
}

[Benchmark]
public int Lookup2()
{
    if (_counts.TryGetValue(_word, out int count))
    {
        return count;
    }

    return -1;
}
```

Method	Mean	Ratio
Lookup1	28.20 ns	1.00
Lookup2	14.12 ns	0.50

Somewhat ironically, even as I write this example, the analyzer and its auto-fixer are helpfully trying to get me to change my benchmark code:



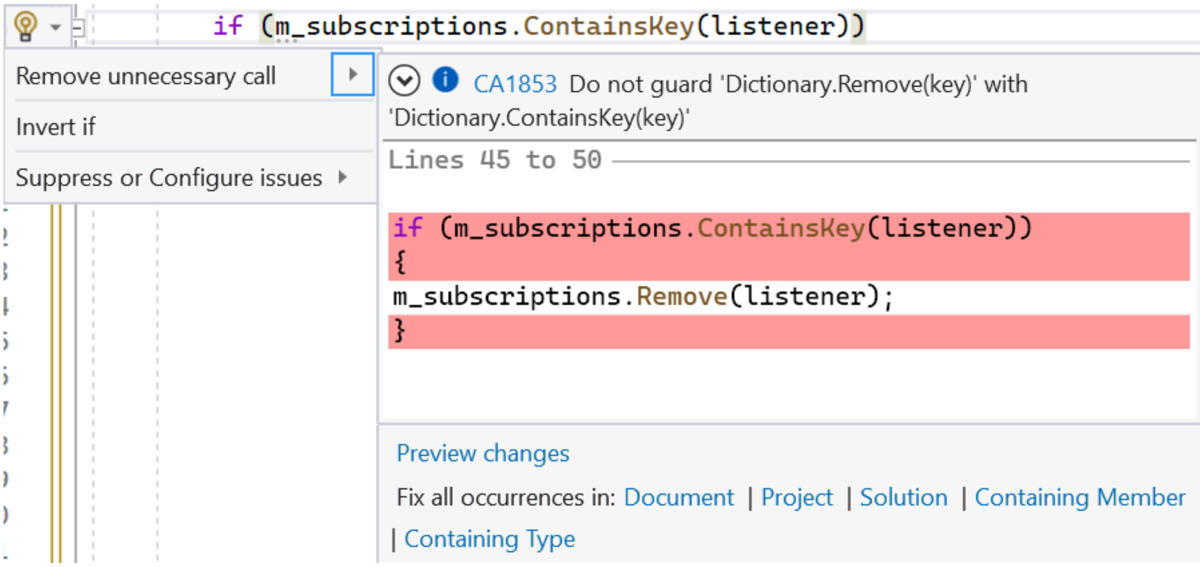
Similarly, [dotnet/roslyn-analyzers#4836]() from [@chucker](https://github.com/chucker) added CA1853, which looks for cases where a `Remove` call on a dictionary is guarded by a `ContainsKey` call. It seems it's fairly natural for developers to only call `Remove` on a dictionary once they're sure the dictionary contains the thing being removed; maybe they think `Remove` will throw an exception if the specified key doesn't exist. However, `Remove` actually allows this as a first-class scenario, with its return `Boolean` value indicating whether the key was in the dictionary (and thus successfully removed) or not. An example of this comes from [dotnet/runtime#68724](), where CA1853 was enabled for dotnet/runtime. The `EventPipeEventDispatcher` type's `RemoveEventListener` method had code like this:

```
if (m_subscriptions.ContainsKey(listener))
{
    m_subscriptions.Remove(listener);
}
```

which the analyzer flagged and which it's auto-fixer replaced with just:

```
m_subscriptions.Remove(listener);
```

Nice and simple. And faster, since as with the TryGetValue case, this is now doing a single dictionary lookup rather than two. :::{custom-style=Figure}



:::

Another nice analyzer added in [dotnet/roslyn-analyzers#5907]() and [dotnet/roslyn-analyzers#5910]() is CA1851, which looks for code that iterates through some kinds of enumerables multiple times. Enumerating an enumerator, whether directly or via helper methods like those in LINQ, can have non-trivial cost. Calling GetEnumerator typically allocates an enumerator object, and every item yielded typically involves two interface calls, one to MoveNext and one to Current. If something can be done via a single pass over the enumerable rather than multiple passes, that can save such costs. In some cases, seeing places this analyzer fires can also inspire changes that avoid any use of enumerators. For example, [dotnet/runtime#67292]() enabled CA1851 for [dotnet/runtime](), and in doing so, it fixed several diagnostics issued by the analyzer (even in a code base that's already fairly stringent about enumerator and LINQ usage). As an example, this is a function in System.ComponentModel.Composition that was flagged by the analyzer:

```
private void InitializeTypeCatalog(IEnumerable<Type> types)
{
    foreach (Type type in types)
    {
        if (type == null)
        {
            throw ExceptionBuilder.CreateContainsNullElement(nameof(types));
        }
        else if (type.Assembly.ReflectionOnly)
        {
            throw new ArgumentException(SR.Format(SR.Argument_ElementReflectionOnlyType,
nameof(types)), nameof(types));
        }
    }

    _types = types.ToArray();
}
```

The method's purpose is to convert the enumerable into an array to be stored, but also to validate that the contents are all non-null and non-"ReflectionOnly." To achieve that, the method is first using a foreach to iterate through the enumerable, validating each element along the way, and then once it's done so, it calls ToArray() to convert the enumerable into an array. There are multiple problems with this. First, it's incurring the expense of interating through the enumerable twice, once for the foreach and once for the `ToArray()`, which internally needs to enumerate it if it can't do something special like cast to `ICollection<Type>` and `CopyTo` the data out of it. Second, it's possible the caller's `IEnumerable<Type>` changes on each iteration, so any validation done in the first iteration isn't actually ensuring there aren't nulls in the resulting array, for example. Since the expectation of the method is that all inputs are valid and we don't need to optimize for the failure cases, the better approach is to *first* call `ToArray()` and then validate the contents of that array, which is exactly what that PR fixes it to do:

```
private void InitializeTypeCatalog(IEnumerable<Type> types)
{
    Type[] arr = types.ToArray();
    foreach (Type type in arr)
    {
        if (type == null)
        {
            throw ExceptionBuilder.CreateContainsNullElement(nameof(types));
        }

        if (type.Assembly.ReflectionOnly)
        {
            throw new ArgumentException(SR.Format(SR.Argument_ElementReflectionOnlyType,
nameof(types)), nameof(types));
        }
    }


    _types = arr;
}
```

With that, we only ever iterate it once (and possibly 0 times if `ToArray` can special-case it, and bonus, we validate on the copy rather than on the mutable original.

Yet another helpful analyzer is the new CA1850 introduced in [dotnet/roslyn-analyzers#4797](https://github.com/dotnet/roslyn-analyzers/pull/4797) from [@wzchua](https://github.com/wzchua). It used to be that if you wanted to cryptographically hash some data in .NET, you would create an instance of a hash algorithm and call its ComputeHash method, e.g.

```
public byte[] Hash(byte[] data)
{
    using (SHA256 h = SHA256.Create())
    {
        return h.ComputeHash(data);
    }
}
```

However, .NET 5 introduced new "one-shot" hashing methods, which obviates the need to create a new `HashAlgorithm` instance, providing a static method that performs the whole operation.

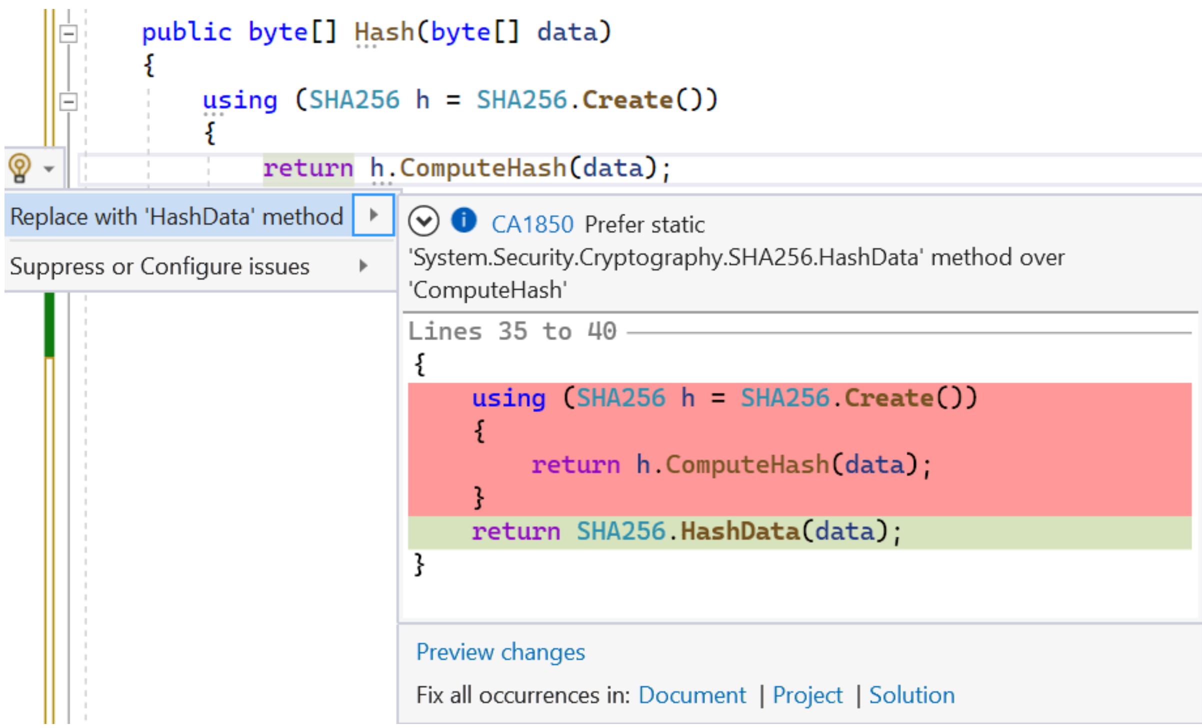
```
public byte[] Hash(byte[] data)
{
    return SHA256.HashData(data);
}
```

CA1850 finds occurrences of the former pattern and recommends changing them to the latter.

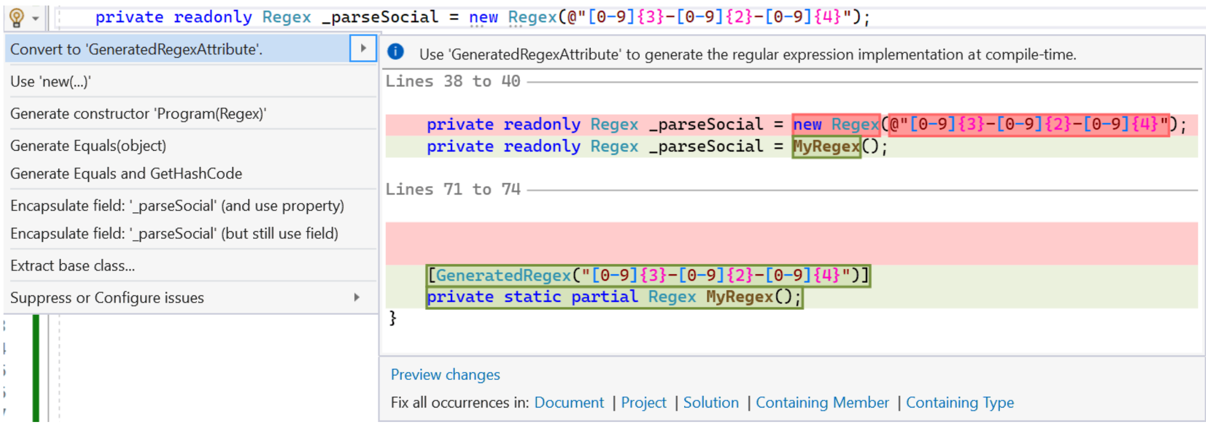
The result is not only simpler, it’s also faster:

```
private readonly byte[] _data = RandomNumberGenerator.GetBytes(128);

[Benchmark(Baseline = true)]
public byte[] Hash1()
{
    using (SHA256 h = SHA256.Create())
    {
        return h.ComputeHash(_data);
    }
}

[Benchmark]
public byte[] Hash2()
{
    return SHA256.HashData(_data);
}
```

Method	Mean	Ratio	Allocated	Alloc Ratio
Hash1	1,212.9 ns	1.00	240 B	1.00
Hash2	950.8 ns	0.78	56 B	0.23



The .NET 7 SDK also includes new analyzers around `[GeneratedRegex(...)]` (dotnet/runtime#68976) and the already mentioned ones for LibraryImport, all of which help to move your code forwards to more modern patterns that have better performance characteristics.



This release also saw [dotnet/runtime](https://github.com/dotnet/runtime) turn on a bunch of additional IDEXXXX code style rules and make a huge number of code changes in response. Most of the resulting changes are purely about simplifying the code, but in almost every case some portion of the changes also have a functional and performance impact.

Let's start with IDE0200, which is about removing unnecessary lambdas. Consider a setup like this:

```csharp
public class C
{
    public void CallSite() => M(i => Work(i));

    public void M(Action<int> action) { }
    private static void Work(int value) { }
}
```

Here we have a method `CallSite` that's invoking a method `M` and passing a lambda to it. Method `M` accepts an `Action<int>`, and the call site is passing a lambda that takes the supplied `Int32` and passes it off to some static functionality. For this code, the C# compiler is going to generate something along the lines of this:

```csharp
public class C
{
    [CompilerGenerated]
    private sealed class <>c
    {
        public static readonly <>c <>9 = new <>c();

        public static Action<int> <>9__0_0;

        internal void <CallSite>b__0_0(int i) => Work(i);
    }

    public void CallSite() => M(<>c.<>9__0_0 ??= new
Action<int>(<>c.<>9.<CallSite>b__0_0));

    public void M(Action<int> action) { }
    private static void Work(int value) { }
}
```

The most important aspect of this is that `<>9__0_0` field the compiler emitted. That field is a cache for the delegate created in `CallSite`. The first time `CallSite` is invoked, it'll allocate a new delegate for the lambda and store it into that field. For all subsequent invocations, however, it'll find the field is non-null and will just reuse the same delegate. Thus, this lambda only ever results in a single allocation for the whole process (ignoring any race conditions on the initial lazy initialization such that multiple threads all racing to initialize the field might end up producing a few additional unnecessary allocations). It's important to recognize this caching only happens because the lambda doesn't access any instance state and doesn't close over any locals; if it did either of those things, such caching wouldn't happen. Secondarily, it's interesting to note the pattern the compiler uses for the lambda itself. Note that generated <CallSite>b__0_0 method is generated as an instance method, and the call site refers to that method of a singleton instance that's used to initialize a <>9 field. That's done because delegates to static methods use something called a "shuffle thunk" to move arguments into the right place for the target method invocation, making delegates to statics ever so slightly more expensive to invoke than delegates to instance methods.

```
private Action _instance = new C().InstanceMethod;
private Action _static = C.StaticMethod;

[Benchmark(Baseline = true)]
public void InvokeInstance() => _instance();

[Benchmark]
public void InvokeStatic() => _static();

private sealed class C
{
    public static void StaticMethod() { }
    public void InstanceMethod() { }
}
```

Method	Mean	Ratio
InvokeInstance	0.8858 ns	1.00
InvokeStatic	1.3979 ns	1.58

So, the compiler is able to cache references to lambdas, great. What about method groups, i.e. where you just name the method directly? Previously, if changed my code to:

```
public class C
{
    public void CallSite() => M(Work);

    public void M(Action<int> action) { }
    private static void Work(int value) { }
}
```

the compiler would generate the equivalent of:

```
public class C
{
    public void CallSite() => M(new Action<int>(Work));

    public void M(Action<int> action) { }
    private static void Work(int value) { }
}
```

which has the unfortunate effect of allocating a new delegate on every invocation, even though we're still dealing with the exact same static method. Thanks to [dotnet/roslyn#58288](https://github.com/dotnet/roslyn/pull/58288) from [@pawchen](https://github.com/pawchen), the compiler will now generate the equivalent of:

```
public class C
{
    [CompilerGenerated]
    private static class <>O
    {
        public static Action<int> <0>__Work;
    }

    public void CallSite() => M(<>O.<0>__Work ??= new Action<int>(Work));

    public void M(Action<int> action) { }
    private static void Work(int value) { }
}
```

Note we again have a caching field that's used to enable allocating the delegate once and caching it. That means that places where code was using a lambda to enable this caching can now switch back to the cleaner and simpler method group way of expressing the desired functionality. There is the interesting difference to be cognizant of that since we don't have a lambda which required the compiler emitting a new method for, we're still creating a delegate directly to the static method. However, the minor difference in thunk overhead is typically made up for by the fact that we don't have a second method to invoke; in the common case where the static helper being invoked isn't inlinable (because it's not super tiny, because it has exception handling, etc.), we previously would have incurred the cost of the delegate invocation plus the non-inlinable method call, and now we just have the cost of an ever-so-slightly more expensive delegate invocation; on the whole, it's typically a wash.

And that brings us to IDE0200, which recognizes lambda expressions that can be removed. dotnet/runtime#71011 enabled the analyzer for dotnet/runtime, resulting in more than 100 call sites changing accordingly. However, IDE0200 does more than just this mostly stylistic change. It also recognizes some patterns that can make a more substantial impact. Consider this code that was changed as part of that PR:

```
Action disposeAction;
IDisposable? disposable = null;
...
if (disposable != null)
{
    disposeAction = () => disposable.Dispose();
}
```

That delegate closes over the `disposable` local, which means this method needs to allocate a display class. But IDE0200 recognizes that instead of closing over `disposable`, we can create the delegate directly to the Dispose method:

```
Action disposeAction;
IDisposable? disposable = null;
...
if (disposable != null)
{
```

```
    disposeAction = disposable.Dispose;
}
```

We still get a delegate allocation, but we avoid the display class allocation, and as a bonus we save on the additional metadata required for the synthesized display class and method generated for the lambda.

IDE0020 is another good example of an analyzer that is primarily focused on making code cleaner, more maintainable, more modern, but that can also lead to removing overhead from many different places. The analyzer looks for code performing unnecessary duplicative casts and recommends using C# pattern matching syntax instead. For example, [dotnet/runtime#70523]() enabled the analyzer and switched more than 250 locations from code like:

```
if (value is SqlDouble)
{
    SqlDouble i = (SqlDouble)value;
    return CompareTo(i);
}
```

to instead be like:

```
if (value is SqlDouble i)
{
    return CompareTo(i);
}
```

In addition to being cleaner, this ends up saving a cast operation, which can add measurable overhead if the JIT is unable to remove it:

```
private object _value = new List<string>();

[Benchmark(Baseline = true)]
public List<string> WithCast()
{
    object value = _value;
    return value is List<string> ? (List<string>)value : null;
}

[Benchmark]
public List<string> WithPattern()
{
    object value = _value;
    return value is List<string> list ? list : null;
}
```

Method	Mean	Ratio
WithCast	2.602 ns	1.00
WithPattern	1.886 ns	0.73

Then there's IDE0031, which promotes using null propagation features of C#. This analyzer typically manifests as recommending changing snippets like:

```
return _value != null ? _value.Property : null;
```

into code that's instead like:

```
return _value?.Property;
```

Nice, concise, and primarily about cleaning up the code and making it simpler and more maintainable by utilizing newer C# syntax. However, there is also a small performance advantage in some situations as well. For example, consider this snippet:

```
public class C
{
    private C _value;

    public int? Get1() => _value != null ? _value.Prop : null;
    public int? Get2() => _value?.Prop;

    public int Prop => 42;
}
```

The C# compiler lowers these expressions to the equivalent of this:

```
public Nullable<int> Get1()
{
    if (_value == null) return null;
    return _value.Prop;
}

public Nullable<int> Get2()
{
    C value = _value;
    if (value == null) return null;
    return value.Prop;
}
```

for which the JIT then generates:

```
; Program.Get1()
       push      rax
       mov       rdx,[rcx+8]
       test      rdx,rdx
       jne       short M00_L00
       xor       eax,eax
       add       rsp,8
       ret
M00_L00:
       cmp       [rdx],dl
       mov       dword ptr [rsp+4],2A
       mov       byte ptr [rsp],1
       mov       rax,[rsp]
       add       rsp,8
       ret
; Total bytes of code 40

; Program.Get2()
       push      rax
       mov       rax,[rcx+8]
       test      rax,rax
       jne       short M00_L00
       xor       eax,eax
       add       rsp,8
       ret
M00_L00:
```

```
        mov       dword ptr [rsp+4],2A
        mov       byte ptr [rsp],1
        mov       rax,[rsp]
        add       rsp,8
        ret
; Total bytes of code 38
```

Note how the `Get1` variant has an extra `cmp` instruction (`cmp [rdx],dl`) in the otherwise identical assembly to `Get2` (other than register selection). That `cmp` instruction in `Get1` is the JIT forcing a null check on the second read of `_value` prior to accessing its `Prop`, whereas in `Get2` the null check against the local means the JIT doesn't need to add an additional null check on the second use of the local, since nothing could have changed it. dotnet/runtime#70965 rolled out additional use of the null propagation operator via auto-fixing IDE0031, resulting in ~120 uses being improved.

Another interesting example is IDE0060, which finds unused parameters and recommends removing them. This was done for non-public members in System.Private.CoreLib in dotnet/runtime#63015. As with some of the other mentioned rules, it's primarily about good hygiene. There can be some small additional cost associated with passing additional parameters (the overhead of reading the values at the call site, putting them into the right register or stack location, etc., and also the metadata size associated with the additional parameter information), but the larger benefit comes from auditing all of the cited violations and finding places where work is simply being performed unnecessarily. For example, that PR made some updates to the `TimeZoneInfo` type's implementation for Unix. In that implementation is a `TZif_ParseRaw` method, which is used to extract some information from a time zone data file. Amongst many input and output parameters, it had `out bool[] StandardTime, out bool[] GmtTime`, which the implementation was dutifully filling in by allocating and populating new arrays for each. The call site for `TZif_ParseRaw` was then taking those arrays and feeding them into another method `TZif_GenerateAdjustmentRules`, which ignored them! Thus, not only was this PR able to remove those parameters from `TZif_GenerateAdjustmentRules`, it was able to update `TZif_ParseRaw` to no longer need to allocate and populate those arrays at all, which obviously yields a much larger gain.

One final example of peanut-buttery performance improvements from applying an analyzer comes from dotnet/runtime#70896 and dotnet/runtime#71361, which applied IDE0029 across dotnet/runtime. IDE0029 flags cases where null coalescing can be used, e.g. flagging:

```
return message != null ? message : string.Empty;
```

and recommending it be converted to:

```
return message ?? string.Empty;
```

As with some of the previous rules discussed, that in and of itself doesn't make a meaningful performance improvement, and rather is about clarity and simplicity. However, in various cases it can. For example, the aforementioned PRs contained an example like:

```
null != foundColumns[i] ? foundColumns[i] : DBNull.Value;
```

which is rewritten to:

```
foundColumns[i] ?? DBNull.Value
```

This avoids an unnecessary re-access to an array. Or again from those PRs the expression:

```
entry.GetKey(_thisCollection) != null ? entry.GetKey(_thisCollection) : "key"
```

being changed to:

```
entry.GetKey(_thisCollection) ?? "key"
```

and avoiding an unnecessary table lookup.

What's Next?

Whew! That was a lot. Congrats on getting through it all.

The next step is on you. Download the latest .NET 7 bits and take them for a spin. Upgrade your apps. Write and share your own benchmarks. Provide feedback, positive and critical. Find something you think can be better? Open an issue, or better yet, submit a PR with the fix. We're excited to work with you to polish .NET 7 to be the best .NET release yet; meanwhile, we're getting going on .NET 8 :)

Until next time...

Happy coding!