Performance Improvements in .NET

Sabato 30 settembre 2023

Mirco Vanini

SPONSORS

altitudo

beanTech

IT moves your business

fluentis

the ERP ready to go live!

[stesi]

Powered by Innovation

Mirco Vanini

Microsoft MVP Developer Technologies

Consultant focused on industrial and embedded solutions using .NET and other native SDKs with over 30 years of experience, XeDotNet community co-founder, speaker and Microsoft MVP since 2012

Microsoft® Most Valuable Professional MVP

@MircoVanini
www.proxsoft.it
https://www.linkedin.com/in/proxsoft

Agenda

- .NET 7 Performance Overview
- JIT – On-Stack Replacement
- Reflection
- String
- Regular Expression
- Native AOT
- JSON
- More & more

.NET 7 Major Themes

- **Performance**
- Simplification & Productivity
- Build Modern Apps
- Cloud-native Apps

.NET 7 Performance Overview

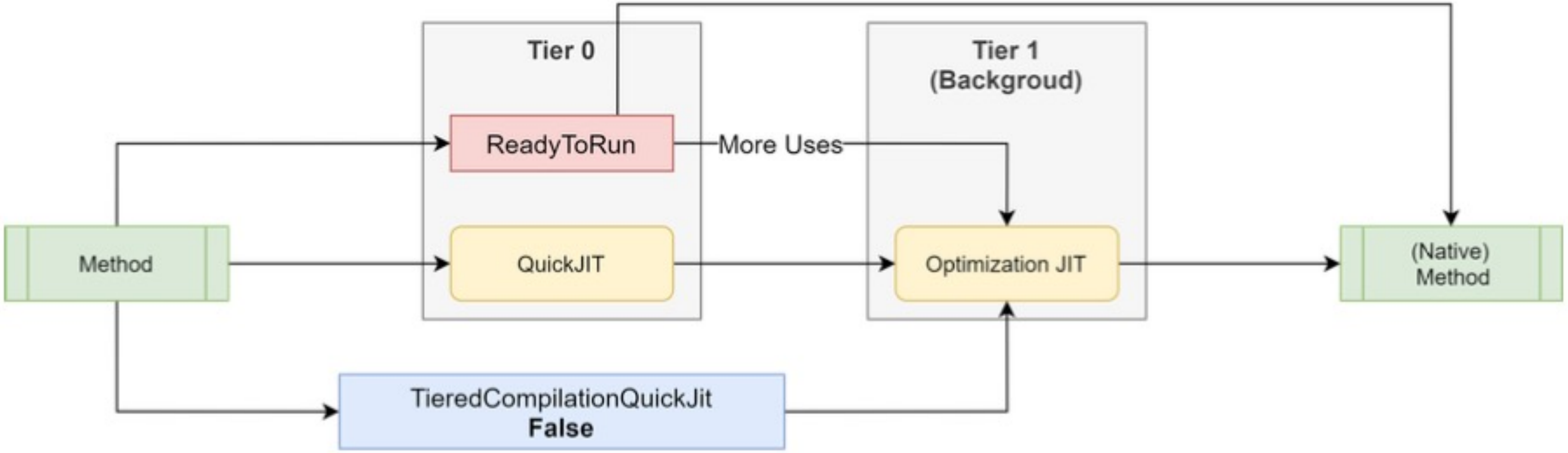
NET 7 – surprise! – is faster than .NET 6, which was faster than .NET 5, which was faster than... well...you get the idea !

Many of the performance features are not specifically driven by the developer, but instead are baked in

Ever since .NET Core hit the scene more than seven years ago, performance has been an integral part of the culture of .NET.

.NET JIT Compilation Details

- JIT Compilation is great...but also has downsides, such as slower start-up speed
- Tiered Compilation is a great compromise between JIT and native code, this has been introduced in .NET Core 3.0 and has been steadily improved since then
- Enable faster startup without sacrificing code quality
- But off by default for methods with loops…



.NET 7 JIT – On-Stack Replacement

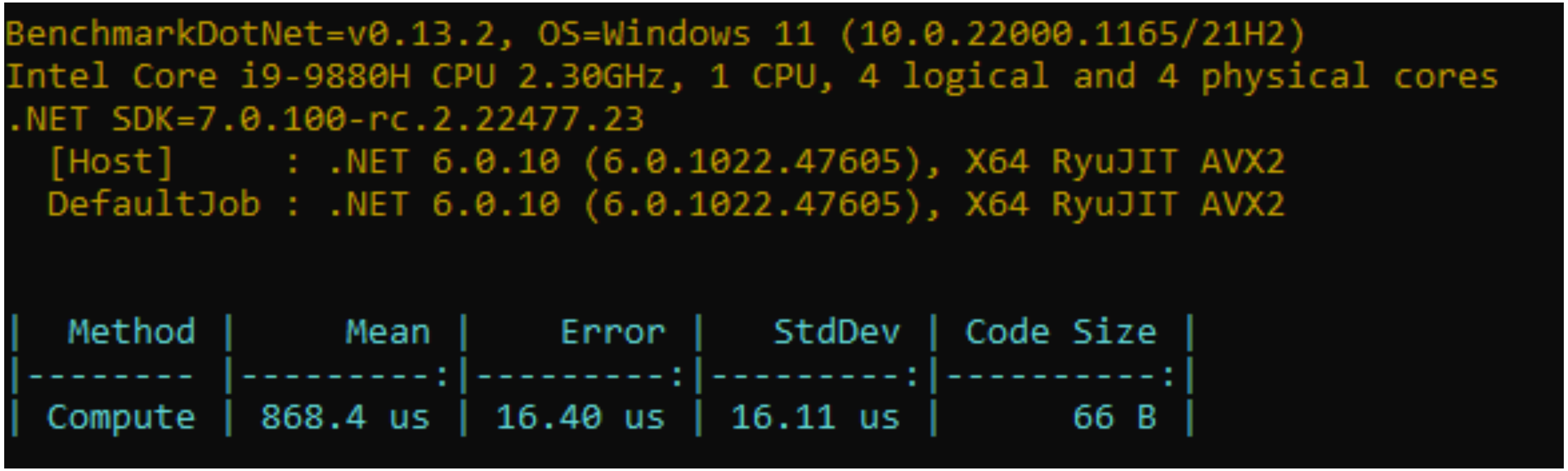
In .NET 7, even methods with loops benefit from tiered compilation. This is achieved via on-stack replacement (OSR).

OSR results in the JIT not only equipping that initial compilation for number of invocations, but also equipping loops for the number of iterations processed.

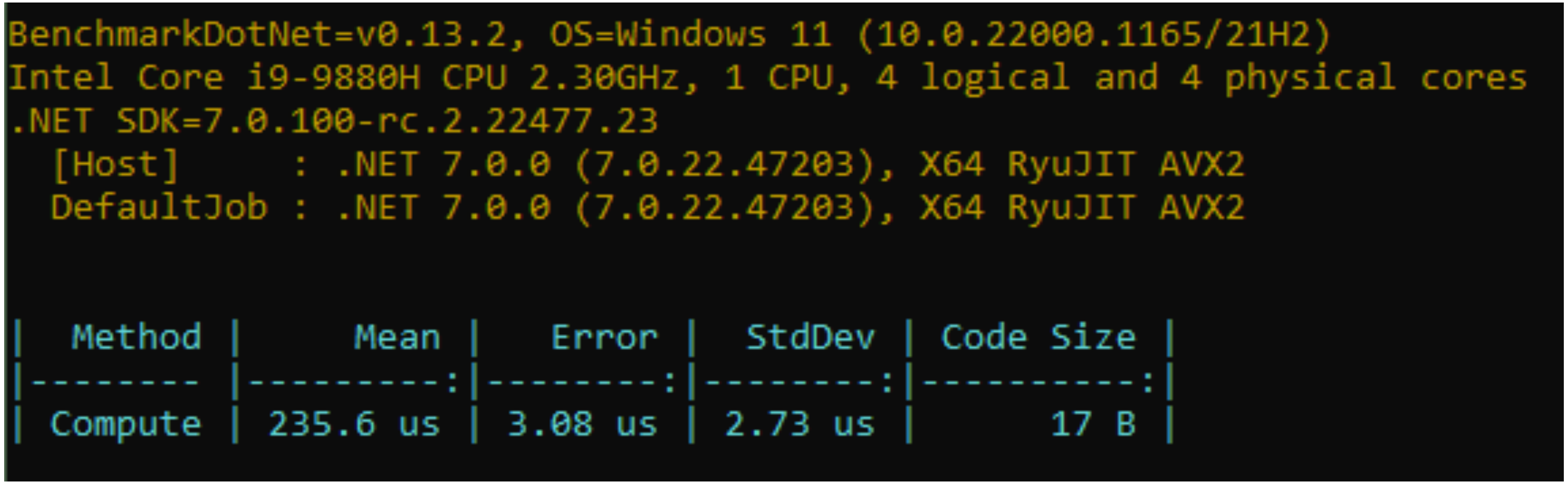
When the number of iterations exceeds a predetermined limit, just as with invocation count, the JIT compiles a new optimized version of the method

DemoOSR / DemoOSRStatic

.NET 7 JIT – On-Stack Replacement



```
BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22000.1165/21H2)
Intel Core i9-9880H CPU 2.30GHz, 1 CPU, 4 logical and 4 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2
  DefaultJob : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2


|  Method |     Mean |     Error |    StdDev | Code Size |
|-------- |---------:|----------:|----------:|----------:|
| Compute | 868.4 us | 16.40 us | 16.11 us |      66 B |
```



```
BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22000.1165/21H2)
Intel Core i9-9880H CPU 2.30GHz, 1 CPU, 4 logical and 4 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 7.0.0 (7.0.22.47203), X64 RyuJIT AVX2
  DefaultJob : .NET 7.0.0 (7.0.22.47203), X64 RyuJIT AVX2


|  Method |     Mean |    Error |   StdDev | Code Size |
|-------- |---------:|---------:|---------:|----------:|
| Compute | 235.6 us | 3.08 us | 2.73 us |      17 B |
```

.NET 7 JIT – Continue…

- PGO (Profile-Guided Optimization)
- Bounds Check Elimination
- Loop Hoisting and Cloning
- Folding, propagation, and substitution
- Vectorization (SIMD)
- Inlining
- Arm64
- JIT helpers
- Grab Bag

.NET 7 Reflection

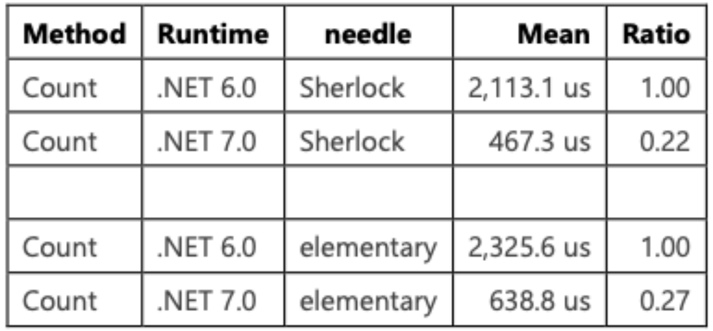
- Reflection invoke has historically had non-trivial overheads
  - Devs worked around that with reflection emit
- Reflection emit approach now built-in

```csharp
private MethodInfo _zeroArgs = typeof(Program).GetMethod(nameof(ZeroArgsMethod));
private MethodInfo _oneArg = typeof(Program).GetMethod(nameof(OneArgMethod));
private object[] _args = new object[] { 42 };

[Benchmark] public void InvokeZero() => _zeroArgs.Invoke(null, null);
[Benchmark] public void InvokeOne() => _oneArg.Invoke(null, _args);

public static void ZeroArgsMethod() { }
public static void OneArgMethod(int i) { }
```

```
|     Method |              Runtime |       Mean |
|----------- |--------------------- |-----------:|
| InvokeZero |             .NET 6.0 |  45.194 ns |
| InvokeZero |             .NET 7.0 |   7.770 ns |
|            |                      |            |
|  InvokeOne |             .NET 6.0 |  82.724 ns |
|  InvokeOne |             .NET 7.0 |  22.758 ns |
```

.NET 7 String

```
// The Project Gutenberg eBook of The Adventures of Sherlock Holmes, by Arthur Conan Doyle
private static readonly string s_haystack = new HttpClient().GetStringAsync("http://aleph.gutenberg.org/1/6/6/1661/1661-0.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;
}
```



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

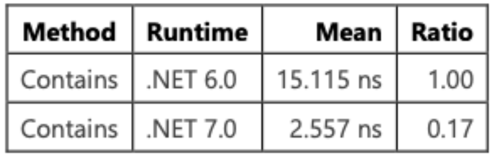
.NET 7 String

```
// The Project Gutenberg eBook of The Adventures of Sherlock Holmes, by Arthur Conan Doyle
private static readonly string s_haystack = new HttpClient().GetStringAsync("http://aleph.gutenberg.org/1/6/6/1661/1661-0.txt").Result;

[Params(StringComparison.Ordinal, StringComparison.OrdinalIgnoreCase)]
public StringComparison Comparison { get; set; }

[Params("elementary")]
public string Needle { get; set; }

[Benchmark]
public int Count()
{
    int count = 0, pos = 0;
    while ((pos = s_haystack.IndexOf(Needle, pos, Comparison)) >= 0)
    {
        pos += Needle.Length;
        count++;
    }

    return count;
}
```

```
| Method |  Runtime |         Comparison |     Needle |         Mean |
|------- |--------- |------------------- |----------- |-------------:|
|  Count | .NET 6.0 |            Ordinal | elementary | 1,064.00 us |
|  Count | .NET 7.0 |            Ordinal | elementary |    57.93 us |
|        |          |                    |            |              |
|  Count | .NET 6.0 | OrdinalIgnoreCase | elementary | 2,332.51 us |
|  Count | .NET 7.0 | OrdinalIgnoreCase | elementary |   631.75 us |
```

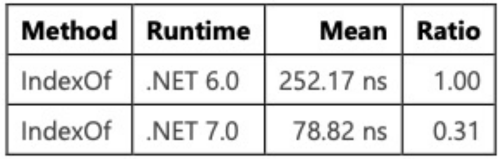
.NET 7 String

```
private byte[] _data = new byte[95];
[Benchmark]
public bool Contains() =>_data.AsSpan().Contains((byte)1);
```



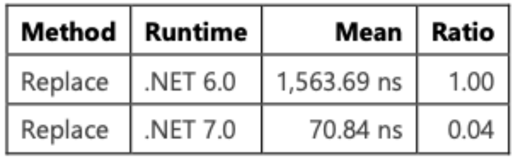
Method	Runtime	Mean	Ratio
Contains	.NET 6.0	15.115 ns	1.00
Contains	.NET 7.0	2.557 ns	0.17

```
private int[] _dataInt = new int[1000];
[Benchmark]
public int IndexOf() => _dataInt.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

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

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

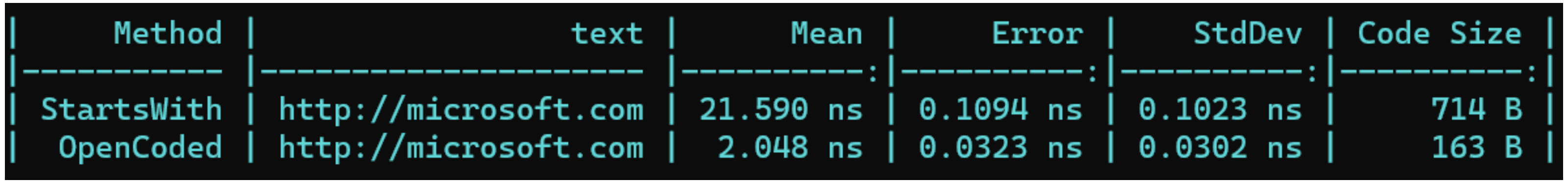


Method	Runtime	Mean	Ratio
Replace	.NET 6.0	1,563.69 ns	1.00
Replace	.NET 7.0	70.84 ns	0.04

.NET 7 String

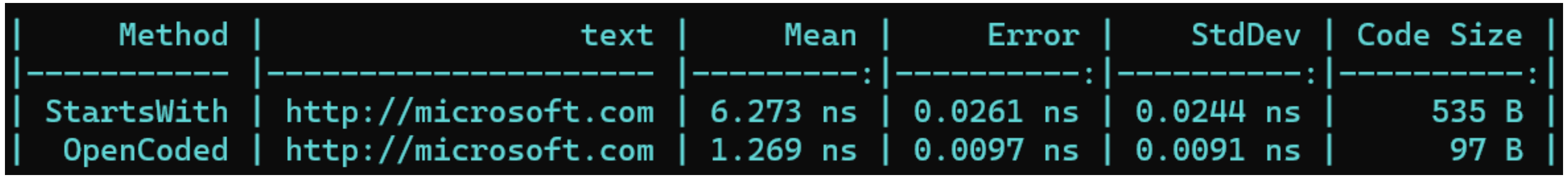
```csharp
[Benchmark]
[Arguments("http://microsoft.com")]
public bool StartsWith(string text) =>
    text.StartsWith("https://",
        StringComparison.OrdinalIgnoreCase);

[Benchmark]
[Arguments("http://microsoft.com")]
public bool OpenCoded(string text) =>
    text.Length >= 8 &&
    (text[0] | 0x20) == 'h' &&
    (text[1] | 0x20) == 't' &&
    (text[2] | 0x20) == 't' &&
    (text[3] | 0x20) == 'p' &&
    (text[4] | 0x20) == 's' &&
    text[5] == ':' &&
    text[6] == '/' &&
    text[7] == '/';
```

.NET Framework 4.8



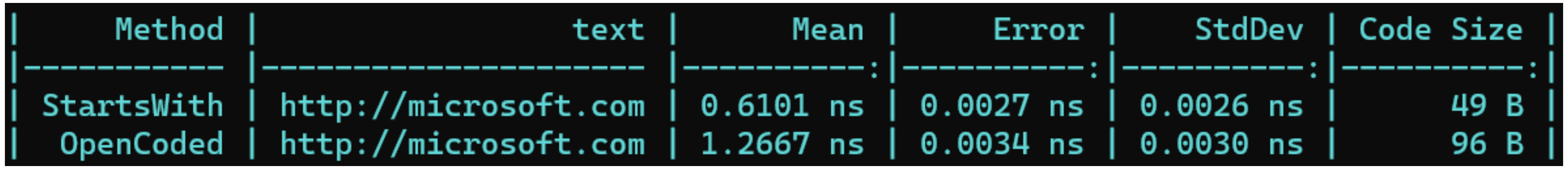
```
|     Method |                 text |      Mean |     Error |    StdDev | Code Size |
|----------- |--------------------- |----------:|----------:|----------:|----------:|
| StartsWith | http://microsoft.com | 21.590 ns | 0.1094 ns | 0.1023 ns |     714 B |
|  OpenCoded | http://microsoft.com |  2.048 ns | 0.0323 ns | 0.0302 ns |     163 B |
```

.NET 6



```
|     Method |                 text |     Mean |     Error |    StdDev | Code Size |
|----------- |--------------------- |---------:|----------:|----------:|----------:|
| StartsWith | http://microsoft.com | 6.273 ns | 0.0261 ns | 0.0244 ns |     535 B |
|  OpenCoded | http://microsoft.com | 1.269 ns | 0.0097 ns | 0.0091 ns |      97 B |
```

.NET 7



```
|     Method |                 text |      Mean |     Error |    StdDev | Code Size |
|----------- |--------------------- |----------:|----------:|----------:|----------:|
| StartsWith | http://microsoft.com | 0.6101 ns | 0.0027 ns | 0.0026 ns |      49 B |
|  OpenCoded | http://microsoft.com | 1.2667 ns | 0.0034 ns | 0.0030 ns |      96 B |
```

Lesson: Work-arounds Should Be Revisited !

.NET 7 Regular Expression

```
private static Regex s_regex = new Regex(@"[a-z]shing", RegexOptions.Compiled);


private static string s_text = new HttpClient().GetStringAsync(@"https://github.com/rust-
leipzig/regex-performance/blob/13915c5182f2662ed906cde557657037c0c0693e/3200.txt").Result;


[Benchmark]
public int SubstringSearch()
{
    int count = 0;
    Match m = s_regex.Match(s_text);
    while (m.Success)
    {
        count++;
        m = m.NextMatch();
    }
    return count;
}
```

```
|          Method |              Runtime |           Mean |  Ratio |
|---------------- |-------------------- |---------------:|-------:|
| SubstringSearch | .NET Framework 4.8 | 3,625.875 us |  1.000 |
| SubstringSearch |             .NET 6.0 |   976.662 us |  0.269 |
| SubstringSearch |             .NET 7.0 |     9.477 us |  0.003 |
```

Non-Prefix String Search

.NET 7 Regular Expression

```
private static Regex s_email = new Regex(@"[\w.+-]+@[\w.-]+.[\w.-]+", RegexOptions.Compiled);


private static string s_text = new
HttpClient().GetStringAsync(@"https://raw.githubusercontent.com/mariomka/regex-
benchmark/8e11300825fc15588e4db510c44890cd4f62e903/input-text.txt").Result;


[Benchmark]
public int Email()
{
    int count = 0;
    Match m = s_email.Match(s_text);
    while (m.Success)
    {
        count++;
        m = m.NextMatch();
    }
    return count;
}
```

```
| Method |             Runtime |             Mean |  Ratio |
|--------|-------------------- |-----------------|--------|
|  Email | .NET Framework 4.8 | 11,019,362.9 us |  1.000 |
|  Email |           .NET 6.0 |     48,723.8 us |  0.048 |
|  Email |           .NET 7.0 |        623.0 us |  0.001 |
```

Literals after loops

.NET 7 Native AOT

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.

Native AOT deployment overview

.NET 7 Native AOT

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.

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

With the JIT, IsDynamicCodeCompiled is true. But with Native AOT, it's false.

.NET 7 Native AOT

.CSPROJ

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

```
dotnet publish -r win-x64 -c Release
dotnet publish -r linux-arm64 -c Release
```

```
<PublishTrimmed>true</PublishTrimmed>
```

```
<InvariantGlobalization>true</InvariantGlobalization>
<DebuggerSupport>false</DebuggerSupport>
<EnableUnsafeUTF7Encoding>false</EnableUnsafeUTF7Encoding>
<EventSourceSupport>false</EventSourceSupport>
<HttpActivityPropagationSupport>false</HttpActivityPropagationSupport>
<InvariantGlobalization>true</InvariantGlobalization>
<MetadataUpdaterSupport>false</MetadataUpdaterSupport>
<UseNativeHttpHandler>true</UseNativeHttpHandler>
<UseSystemResourceKeys>true</UseSystemResourceKeys>
```

Trimming Options

.NET 7 ReadyToRun

```
XYZ.Business.dll     -   487.424 bytes
XYZ.Business.r2r.dll - 9.850.880 bytes
```

```
dotnet publish
      --self-contained
      -r linux-arm
      -p:PublishReadyToRunComposite=true
      -p:PublishReadyToRun=true
      -p:PublishDir=${workspaceFolder}/Deploy/linux-arm/publish
      -c Release
      /maxcpucount:1
```

R2R binaries improve startup performance by reducing the amount of work the just-in-time (JIT) compiler needs to do as your application loads. The binaries contain similar native code compared to what the JIT would produce. However, R2R binaries are larger because they contain both intermediate language (IL) code, which is still needed for some scenarios, and the native version of the same code.

Use of Composite ReadyToRun is only recommended for applications that disable Tiered Compilation or applications running on Linux that are seeking the best startup time with self-contained deployment.

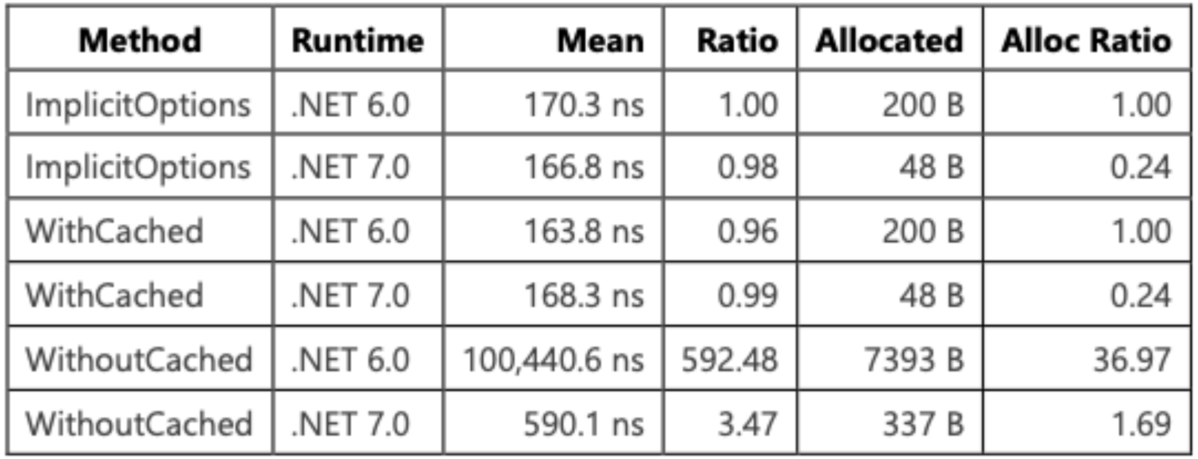
.NET 7 JSON

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.

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

.NET 7 JSON

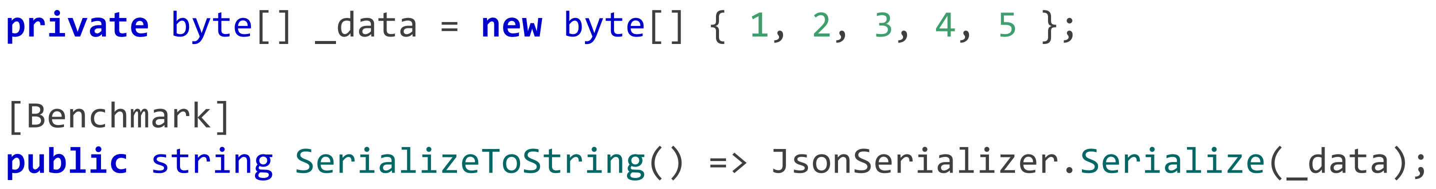
```
private JsonSerializerOptions _options = new JsonSerializerOptions();
private MyAmazingClass _instance = new MyAmazingClass();

[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; }
}
```



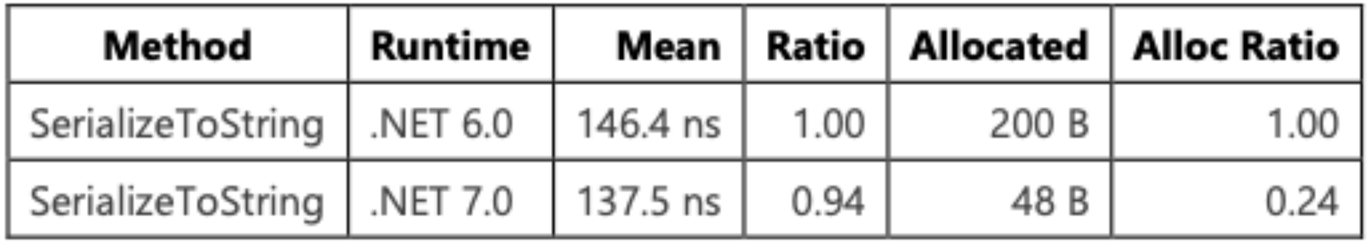
Method	Runtime	Mean	Ratio	Allocated	Alloc Ratio
ImplicitOptions	.NET 6.0	170.3 ns	1.00	200 B	1.00
ImplicitOptions	.NET 7.0	166.8 ns	0.98	48 B	0.24
WithCached	.NET 6.0	163.8 ns	0.96	200 B	1.00
WithCached	.NET 7.0	168.3 ns	0.99	48 B	0.24
WithoutCached	.NET 6.0	100,440.6 ns	592.48	7393 B	36.97
WithoutCached	.NET 7.0	590.1 ns	3.47	337 B	1.69

.NET 7 JSON

Utf8JsonWriter and Utf8JsonReader also saw several improvements directly. (CopyString method which provides a non-allocating mechanism to get access to a string value from the reader).



```
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

.NET 7 MS research is on-going

Symbolic Regex Matcher

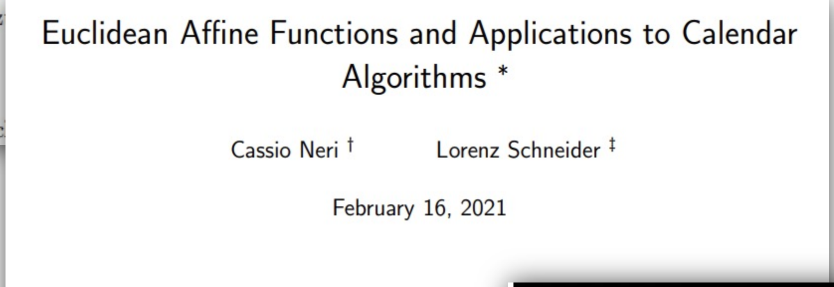
Olli Saarikivi[1], Margus Veanes[1], Tiki Wan[2], and Eric Xu[2]

[1] Microsoft Research, Redmond, USA
[2] Microsoft Az

Abstract. Symbolic regex matc

`RegexOptions.NonBacktracking`

[Regular Expression Improvements in .NET 7]



Euclidean Affine Functions and Applications to Calendar Algorithms *

Cassio Neri † Lorenz Schneider ‡

February 16, 2021

Abstract

We study properties of Euclidean affine functions (EAFs), nan
and their closely related expression $\mathring{f}(r) = (\alpha \cdot r + \beta)\%\delta$, where r
% respectively denote the quotient and remainder of Euclidean d
numerical approximations that are important for the efficient e

`DateTime.Month/Day/Year`

[Euclidean Affine Functions and Applications to Calendar]

Floating-point Number Parsing With Perfect Accuracy at a Gigabyte Per Second

Daniel Lemire

Computer Science Professor

GO SYSTEMS CONF

`double/float.{Try}Parse`

[Number Parsing at a Gigabyte per Second]

.NET 7 More & More & More…

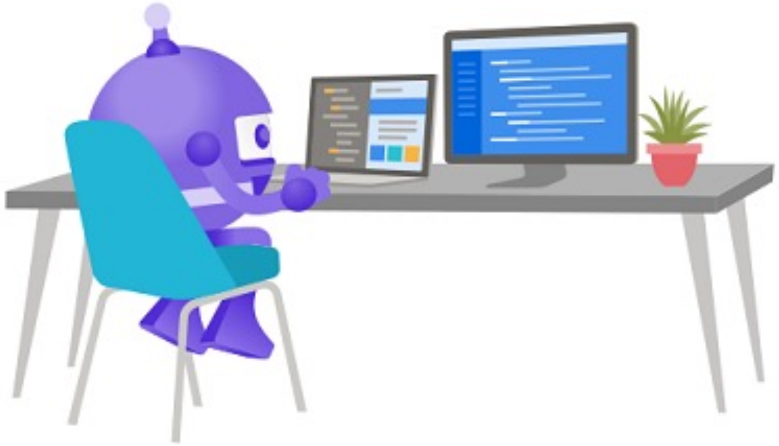
- GC (Area)
- Reflection
- Interop
- Threading
- Primitive Types and Numerics
- Regex
- Collection
- LINQ
- File I/O
- Compression
- Networking
- Mono
- XML
- Cryptography
- Diagnostics
- Exceptions
- Registry
- Analyzer

Performance Improvements in .NET 7

Over 500 PRs / improvements discussed (> 20% from outside of the .NET team, yay open source!)

Thanks!

Saturday Innova community



.NET

your platform for building anything

Sabato 30 settembre 2023

Mirco Vanini