# Performance Improvements in .NET 9

Mirco Vanini

## Sponsor

CONSORZIO UNIVERSITARIO DI PORDENONE
MOLTIPLICATORE DI VALORE

altitudo

beanTech
IT moves your business

[stesi]
Powered by Innovation

***"Programs don't run on mathematical abstractions, they run on hardware"***

**Bjarne Stroustrup**

# .NET Performance Improvement Journey – JIT Compiler

**.NET Core 2.0**
August 2017
RyuJIT Introduction
Devirtualization

Tiered Compilation
Span<T>
May 2018
**.NET Core 2.1**

**.NET Core 3.0**
September 2019
Hardware Intrinsics
Improved Inlining

Profile PGO
Crossgen2
November 2020
**.NET 5**

**.NET 6**
November 2021
Dynamic PGO
Vectorization

Native AOT
Enhanced SIMD
November 2022
**.NET 7**

**.NET 8**
November 2023
Bounds Check Elimination
Inlining and Devirtualization

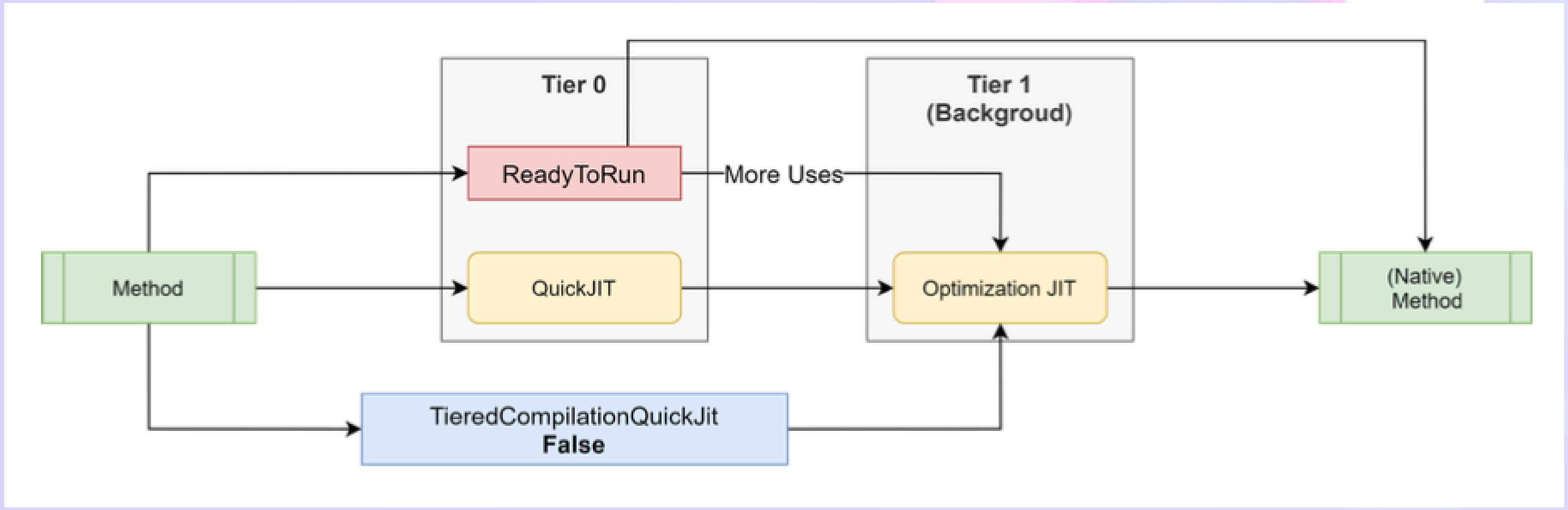
New Architectures
SIMD and Vectorization
Arm64
November 2024
**.NET 9**

**The .NET Performance Improvement Journey is a continuous process aimed at enhancing the efficiency, speed, and overall performance of .NET application**

# Tiered Compilation

Tiered compilation in the Just-In Time (JIT) compiler for C# is a performance optimization technique that allows the JIT compiler to compile methods in multiple stages, or "tiers," to balance startup time and runtime performance.

- Quick JIT (Tier 0): When a method is first called, it is compiled quickly with minimal optimizations. This allows the application to start and run faster initially.
- Optimized JIT (Tier 1): If the method is called frequently, the JIT compiler recompiles it with more aggressive optimizations. This tier aims to improve the performance of hot paths in the code.



# Static Profile Guided Optimization (PGO)

Static Profile Guided Optimization (PGO) is a technique used to optimize the performance of applications by using profiling data collected during a representative run of the application.

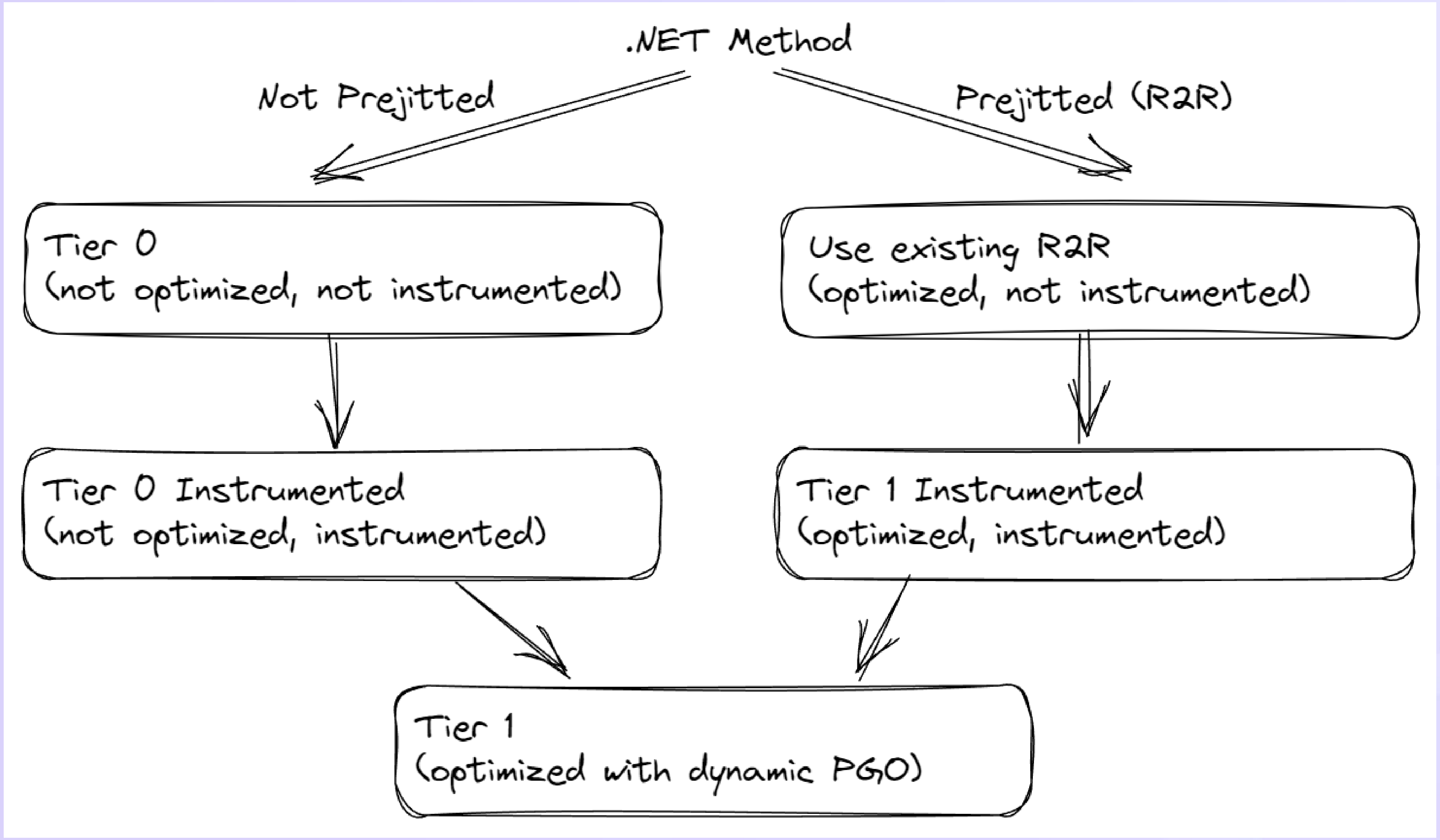
- Profiling Phase: The application is run with typical workloads to collect detailed information about its execution. This includes data on which methods are called most frequently, the paths taken through the code, and other runtime behaviours.
- Profile Data Analysis: The collected profile data is analysed to identify performance critical areas of the code.
- Optimization Phase: The compiler uses the profile data to apply targeted optimizations during the build process. This can include inlining frequently called methods, optimizing hot paths, and improving branch prediction

# Dynamic Profile Guided Optimization (PGO)

Dynamic Profile Guided Optimization (PGO) is a technique that optimizes the performance of an application by collecting and using runtime data to guide the optimization process, it collects data while the application is running and uses it to make real-time optimizations.

- Data Collection: As the application runs, the runtime collects detailed information about its execution. This includes data on which methods are called most frequently, the paths taken throu gh the code, and other runtime behaviours.
- Analysis: The collected data is analysed to identify performance-critical areas of the code.
- Optimization: The JIT compiler uses the runtime data to apply targeted optimizations. This can include inlining frequently called methods, optimizing hot paths, and improving branch prediction

# JIT – PGO (Profile Guided Optimization)


.NET Method
Not Prejitted
Prejitted (R2R)
Tier 0
(not optimized, not instrumented)
Use existing R2R
(optimized, not instrumented)
Tier 0 Instrumented
(not optimized, instrumented)
Tier 1 Instrumented
(optimized, instrumented)
Tier 1
(optimized with dynamic PGO)

## Demo

## PGO - Type checks and casts

Dynamic PGO is now able to track the most common input types to cast operations and then when generating the optimized code, emit special checks that add fast paths for the most common types

```
public class A { }
public class B : A { }
public class C : B { }
private A _obj = new C();

[Benchmark]
public bool DemoCast() => _obj is B;
```

```
| Method   | Runtime  | Mean      | Ratio | Rank | Code Size |
|--------- |--------- |----------:|------:|-----:|----------:|
| DemoCast | .NET 8.0 | 3.5395 ns |  1.00 |    2 |      38 B |
| DemoCast | .NET 9.0 | 0.7902 ns |  0.21 |    1 |      65 B |
```

# PGO - Type checks and casts

Disassembly on .NET 8

```
push       rax
mov        rsi,[rdi+8]
mov        rdi,offset MT_DemoCast+B
call       qword ptr [7F3D91524360]
           ; System.Runtime.CompilerServices.CastHelpers.IsInstanceOfClass(Void*, System.Object)
test       rax,rax
setne      al
movzx      eax,al
add        rsp,8
ret

; Total bytes of code 35
```

# PGO - Type checks and casts

Disassembly on .NET 9

```
       push      rbp
       mov       rbp,rsp
       mov       rsi,[rdi+8]
       mov       rcx,rsi
       test      rcx,rcx
       je        short M00_L00
       mov       rax,offset MT_DemoCast+C
       cmp       [rcx],rax
       jne       short M00_L01
M00_L00:
       test      rcx,rcx
       setne     al
       movzx     eax,al
       pop       rbp
       ret
M00_L01:
       mov       rdi,offset MT_DemoCast+B
       call      System.Runtime.CompilerServices.CastHelpers.IsInstanceOfClass(Void*, System.Object)
       mov       rcx,rax
       jmp       short M00_L00

; Total bytes of code 62
```

On .NET 8, it's loading the reference to the object and the desired method token for  and calling the JIT helper to do the type check.

On .NET 9, instead it's loading the method token for, which it saw during profiling to be the most common type used and then comparing that against the actual object's method token.

# PGO - Profiled Equal / SequenceEqual

```
[Benchmark]
[Arguments("abcd", "abcg")]
public bool DemoEquals(string a, string b) => a == b;
```

```
| Method      | Runtime   | a    | b    | Mean     | Ratio | Rank | Code Size |
|------------ |---------- |----- |----- |---------:|------:|-----:|----------:|
| DemoEquals | .NET 8.0 | abcd | abcg | 3.695 ns |  1.00 |    2 |      76 B |
| DemoEquals | .NET 9.0 | abcd | abcg | 3.203 ns |  0.87 |    1 |     105 B |
```

```
static int[] _dataLeft  = [1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1];
static int[] _dataRight = [1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1];
[Benchmark]
public bool DemoSequenceEquals() => _dataLeft.AsSpan().SequenceEqual(_dataRight);
```

```
| Method              | Runtime   | Mean     | Ratio | Rank | Code Size |
|-------------------- |---------- |---------:|------:|-----:|----------:|
| DemoSequenceEquals | .NET 8.0 | 5.274 ns |  1.00 |    2 |      97 B |
| DemoSequenceEquals | .NET 9.0 | 3.578 ns |  0.68 |    1 |      97 B |
```

It optimize Buffer.Memmove (which is the workhorse behind methods like Span<T>.CopyTo) and SpanHelpers.SequenceEqual (which is the implementation behind methods like string.Equals)

# Loop

```
[Benchmark]
public int DemoLoop()
{
   int sum = 0;
   for (int i = 0; i < 1024; i++)
      sum += i;

     return sum;
}
```

```
| Method    | Runtime   | Mean      | Ratio | Rank | Code Size |
|---------- |---------- |----------:|------:|-----:|----------:|
| DemoLoop  | .NET 8.0  | 305.0 ns  |  1.00 |    2 |      17 B |
| DemoLoop  | .NET 9.0  | 265.5 ns  |  0.88 |    1 |      17 B |
```

- Induction-variable widening
- Post-indexed addressing
- Strength reduction
- Loop counter variable direction

## Tier 0

```
[Benchmark]
public void DemoT0()
{
   for (int i = 0; i < 1024; i++)
      ThrowIfNull(i);
}

private void ThrowIfNull<T>(T a)
{
   ArgumentNullException.ThrowIfNull(a);
}
```

```
| Method | Runtime  | Mean     | Ratio | Rank | Code Size |
|------- |--------- |---------:|------:|-----:|----------:|
| DemoT0 | .NET 8.0 | 605.8 ns |  1.00 |    2 |      12 B |
| DemoT0 | .NET 9.0 | 451.1 ns |  0.80 |    1 |      10 B |
```

# Object stack allocation for boxes

```
[Benchmark]
public int DemoObjectStackAllocationForBoxes()
{
    bool result = Compare(3, 4);
    return result ? 0 : 100;
}

bool Compare(object? x, object? y)
{
    if ((x == null) || (y == null))
    {
        return x == y;
    }

    return x.Equals(y);
}
```

```
| Method                            | Runtime  | Mean       | Ratio | Rank | Code Size |
|---------------------------------- |--------- |-----------:|------:|-----:|----------:|
| DemoObjectStackAllocationForBoxes | .NET 8.0 | 11.3887 ns |  1.00 |    2 |      85 B |
| DemoObjectStackAllocationForBoxes | .NET 9.0 |  0.6959 ns |  0.06 |    1 |       6 B |
```

# Bounds Checks

```
[Benchmark]
[Arguments(3)]
public int DemoBoundsChecks() => Calculate(0, "1234567890abcdefghijklmnopqrstuvwxyz");

[MethodImpl(MethodImplOptions.NoInlining)]
public static int Calculate(int i, ReadOnlySpan<char> src)
{
   int sum = 0;

   for (; (uint)i < src.Length; i++)
   {
      sum += src[i];
   }

   return sum;
}
```

```
| Method             | Runtime  | Mean     | Ratio | Rank | Code Size |
|------------------- |--------- |---------:|------:|-----:|----------:|
| DemoBoundsChecks   | .NET 8.0 | 38.94 ns |  1.00 |    2 |      53 B |
| DemoBoundsChecks   | .NET 9.0 | 30.09 ns |  0.78 |    1 |      53 B |
```

# Bounds Checks

```
private readonly string[] _names = Enum.GetNames<MyEnum>();
public enum MyEnum : ulong { A, B, C, D }

[Benchmark]
[Arguments(2)]
public string? DemoBoundsChecks2(ulong ulValue)
{
    string[] names = _names;
    string? ret = null;

    for(int i = 0; i < 1024; i++)
    {
        ret = ulValue < (ulong)names.Length ?
                    names[(uint)ulValue] :
                    null;
    }

    return ret;
}
```

```
| Method             | Runtime   | ulValue | Mean         | Ratio | Rank | Code Size |
|------------------- |---------- |-------- |------------:|------:|-----:|----------:|
| DemoBoundsChecks2 | .NET 8.0  | 2       | 1,152.9 ns | 1.00 |    2 |      79 B |
| DemoBoundsChecks2 | .NET 9.0  | 2       |   580.9 ns | 0.56 |    1 |      40 B |
```

# ARM64

.NET 9 brings several enhancements for ARM64

- **Scalable Vector Extension (SVE)**: .NET 9 introduces experimental support for SVE, a SIMD instruction set for ARM64 CPUs. This allows applications to leverage 128-bit vector registers on NEON-capable hardware.
- **Vector Construction**: Improved vector construction by enabling the mono JIT to utilize the ARM64 ins (Insert) instruction when creating one float or double vector from the values in another.
- **Loop Optimizations**: Enhanced loop optimizations to improve performance in iterative code.
- **Inlining**: Improved inlining capabilities to reduce method call overhead and enhance execution speed.
- **Code Generation**: Enhanced ARM64 vectorization and code generation for better performance..

**Making .NET on Arm an awesome and fast experience has been a critical, multi-year investment.**

## AVX10v1 support

- New APIs have been added for AVX10, which is a new SIMD instruction set from Intel. You can accelerate your .NET applications on AVX10-enabled hardware with vectorized operations using the new Avx10v1 APIs.

# AVX512

```
[Benchmark]

public Vector512<byte> DemoVector512() => Exp512(default, default, default);


[MethodImpl(MethodImplOptions.NoInlining)]

public static Vector512<byte> Exp512(Vector512<byte> a, Vector512<byte> b, Vector512<byte> c) =>

    Vector512.ConditionalSelect(a, b + c, b - c);
```

```
| Method         | Runtime  | Mean     | Ratio | Rank | Code Size |
|--------------- |--------- |---------:|------:|-----:|----------:|
| DemoVector512 | .NET 8.0 | 7.459 ns |  1.00 |    2 |     102 B |
| DemoVector512 | .NET 9.0 | 5.026 ns |  0.68 |    1 |      99 B |
```

```
[Benchmark]
public void DemoVector512Bis() => Vector512.Create("0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef"u8);
```

```
| Method         | Runtime  | Mean     | Ratio | Rank | Code Size |
|--------------- |--------- |---------:|------:|-----:|----------:|
| DemoVector512 | .NET 8.0 | 7.459 ns |  1.00 |    2 |     102 B |
| DemoVector512 | .NET 9.0 | 5.026 ns |  0.68 |    1 |      99 B |
```

# Vectorization

```
private Vector128<byte> _v1 = Vector128.Create((byte)0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);

[Benchmark]
public Vector128<byte> DemoVectorSquare() => _v1 * _v1;
```

```
| Method           | Runtime  | Mean       | Ratio | Rank | Code Size |
|----------------- |--------- |-----------:|------:|-----:|----------:|
| DemoVectorSquare | .NET 8.0 | 34.8638 ns | 1.000 |    2 |     181 B |
| DemoVectorSquare | .NET 9.0 |  0.2373 ns | 0.008 |    1 |      39 B |
```

```
private byte[] _dataToHash = new byte[1024 * 1024];

[GlobalSetup]
public void Setup() => new Random(42).NextBytes(_dataToHash);

[Benchmark]
public UInt128 DemoVectorHash() => XxHash128.HashToUInt128(_dataToHash);
```

```
| Method         | Runtime  | Mean      | Ratio | Rank | Code Size |
|--------------- |--------- |----------:|------:|-----:|----------:|
| DemoVectorHash | .NET 8.0 | 144.72 us |  1.00 |    2 |     163 B |
| DemoVectorHash | .NET 9.0 |  99.57 us |  0.69 |    1 |     163 B |
```

# Object Stack Allocation

```
[Benchmark]
public int DemoObjectStackAllocation() => new MyObj(42).Value;


private class MyObj
{
   public MyObj(int value) => Value = value;
   public int Value { get; }
}
```

```
| Method                    | Runtime  | Mean      | Ratio | Rank | Code Size | Allocated | Alloc Ratio |
|-------------------------- |--------- |----------:|------:|-----:|----------:|----------:|------------:|
| DemoObjectStackAllocation | .NET 8.0 | 5.3991 ns |  1.00 |    2 |      34 B |      24 B |        1.00 |
| DemoObjectStackAllocation | .NET 9.0 | 0.0697 ns |  0.01 |    1 |       6 B |         - |        0.00 |
```

**For years, .NET has explored the possibility of stack-allocating managed objects,  In .NET 9, object stack allocation starts to happen. Before you get too excited, it's limited in scope right now !**

## Faster exceptions

- The CoreCLR runtime has adopted a new exception handling approach that improves the performance of exception handling. The new implementation is based on the NativeAOT runtime's exception-handling model. The change removes support for Windows structured exception handling (SEH) and its emulation on Unix. The new approach is supported in all environment except for Windows x86 (32-bit).
- The new exception handling implementation is 2-4 times faster
- The new implementation is enabled by default
  - Set `System.Runtime.LegacyExceptionHandling` to true in the runtimeconfig.json file.
  - Set the `DOTNET_LegacyExceptionHandling` environment variable to 1.

# Span, Span, and more Span

```csharp
[Benchmark]
public string DemoSpan() => Path.Join("a", "b", "c", "d", "e");
```

```
| Method   | Runtime  | Mean      | Ratio | Rank | Code Size | Allocated | Alloc Ratio |
|--------- |--------- |----------:|------:|-----:|----------:|----------:|------------:|
| DemoSpan | .NET 8.0 | 115.02 ns |  1.00 |    2 |     107 B |     104 B |        1.00 |
| DemoSpan | .NET 9.0 |  80.03 ns |  0.70 |    1 |     137 B |      40 B |        0.38 |
```

```csharp
[Benchmark]
[Arguments("helloworld.txt")]
public bool DemoSpan2(string path) => path.EndsWith(".txt", StringComparison.OrdinalIgnoreCase);
```

```
| Method    | Runtime  | path           | Mean      | Ratio | Rank | Code Size | Allocated | Alloc Ratio |
|---------- |--------- |--------------- |----------:|------:|-----:|----------:|----------:|------------:|
| DemoSpan2 | .NET 8.0 | helloworld.txt | 10.733 ns |  1.00 |    2 |      27 B |         - |          NA |
| DemoSpan2 | .NET 9.0 | helloworld.txt |  1.832 ns |  0.21 |    1 |      56 B |         - |          NA |
```

**The introduction of and back in .NET Core 2.1 have revolutionized how we write .NET code (especially in the core libraries)**
**What is moral? Span, Span, and more Span !**

# LINQ

```
private IEnumerable<int> _arrayDistinct = Enumerable.Range(0, 1000).ToArray().Distinct();
private IEnumerable<int> _appendSelect = Enumerable.Range(0, 1000).ToArray().Append(42).Select(i => i * 2);
private IEnumerable<int> _rangeReverse = Enumerable.Range(0, 1000).Reverse();
private IEnumerable<int> _listDefaultIfEmptySelect =
                                        Enumerable.Range(0, 1000).ToList().DefaultIfEmpty().Select(i => i * 2);
private IEnumerable<int> _listSkipTake = Enumerable.Range(0, 1000).ToList().Skip(500).Take(100);
private IEnumerable<int> _rangeUnion = Enumerable.Range(0, 1000).Union(Enumerable.Range(500, 1000));
```

```
[Benchmark] public int DemoDistinctFirst()                          => _arrayDistinct.First();
[Benchmark] public int DemoAppendSelectLast()              => _appendSelect.Last();
[Benchmark] public int DemoRangeReverseCount()                      => _rangeReverse.Count();
[Benchmark] public int DemoDefaultIfEmptySelectElementAt() => _listDefaultIfEmptySelect.ElementAt(999);
[Benchmark] public int DemoListSkipTakeElementAt()                  => _listSkipTake.ElementAt(99);
[Benchmark] public int DemoRangeUnionFirst()               => _rangeUnion.First();
```

## LINQ

```
| Method                        | Runtime  | Mean           | Ratio   | Allocated | Alloc Ratio |
|------------------------------:|--------:|---------------:|--------:|----------:|------------:|
| DemoDistinctFirst             | .NET 8.0 |     185.274 ns |  1.00   |     328 B |        1.00 |
| DemoDistinctFirst             | .NET 9.0 |       8.933 ns |  0.05   |         - |        0.00 |
|                               |          |                |         |           |             |
| AppendSelectLast              | .NET 8.0 |   3,668.347 ns |  1.000  |     144 B |        1.00 |
| AppendSelectLast              | .NET 9.0 |       2.222 ns |  0.001  |         - |        0.00 |
|                               |          |                |         |           |             |
| RangeReverseCount             | .NET 8.0 |       8.703 ns |  1.00   |         - |          NA |
| RangeReverseCount             | .NET 9.0 |       3.465 ns |  0.40   |         - |          NA |
|                               |          |                |         |           |             |
| DefaultIfEmptySelectElementAt | .NET 8.0 |   2,772.283 ns |  1.000  |     144 B |        1.00 |
| DefaultIfEmptySelectElementAt | .NET 9.0 |       4.399 ns |  0.002  |         - |        0.00 |
|                               |          |                |         |           |             |
| ListSkipTakeElementAt         | .NET 8.0 |       3.699 ns |  1.00   |         - |          NA |
| ListSkipTakeElementAt         | .NET 9.0 |       2.103 ns |  0.57   |         - |          NA |
|                               |          |                |         |           |             |
| RangeUnionFirst               | .NET 8.0 |      53.670 ns |  1.00   |     344 B |        1.00 |
| RangeUnionFirst               | .NET 9.0 |       5.181 ns |  0.10   |         - |        0.00 |
```

# LINQ

```
private string[] _values = [];


[Benchmark] public object DemoChunk()       => _values.Chunk(10);
[Benchmark] public object DemoDistinct()    => _values.Distinct();
[Benchmark] public object DemoJoin()        => _values.Join(_values, i => i, i => i, (i,j) => i);
[Benchmark] public object DemoToLookup()    => _values.ToLookup(i => i);
[Benchmark] public object DemoReverse()     => _values.Reverse();
[Benchmark] public object DemoSelectIndex() => _values.Select((s, i) => i);
[Benchmark] public object DemoSelectMany()  => _values.SelectMany(i => i);
[Benchmark] public object DemoSkipWhile()   => _values.SkipWhile(i => true);
[Benchmark] public object DemoTakeWhile()   => _values.TakeWhile(i => true);
[Benchmark] public object DemoWhereIndex()  => _values.Where((s, i) => true);
[Benchmark] public object DemoGroupJoin() =>_values.GroupJoin(_values, i => i, i => i, (i,j) => i);
```

## LINQ

```
| Method            | Runtime  | Mean          | Ratio    | Allocated  | Alloc Ratio  |
|------------------:|---------:|--------------:|---------:|-----------:|-------------:|
| Chunk             | .NET 8.0 |   10.7213 ns |     1.00 |       72 B |         1.00 |
| Chunk             | .NET 9.0 |    4.1320 ns |     0.39 |          - |         0.00 |
|                   |          |               |          |            |              |
| Distinct          | .NET 8.0 |    9.4410 ns |     1.00 |       64 B |         1.00 |
| Distinct          | .NET 9.0 |    0.7162 ns |     0.08 |          - |         0.00 |
|                   |          |               |          |            |              |
| GroupJoin         | .NET 8.0 |   22.4746 ns |     1.00 |      144 B |         1.00 |
| GroupJoin         | .NET 9.0 |    1.1356 ns |     0.05 |          - |         0.00 |
|                   |          |               |          |            |              |
| Join              | .NET 8.0 |   18.6332 ns |     1.00 |      168 B |         1.00 |
| Join              | .NET 9.0 |    1.3585 ns |     0.07 |          - |         0.00 |
|                   |          |               |          |            |              |
| ToLookup          | .NET 8.0 |   23.3518 ns |     1.00 |      128 B |         1.00 |
| ToLookup          | .NET 9.0 |    0.9539 ns |     0.04 |          - |         0.00 |
|                   |          |               |          |            |              |
| Reverse           | .NET 8.0 |    9.5791 ns |     1.00 |       48 B |         1.00 |
| Reverse           | .NET 9.0 |    0.9947 ns |     0.10 |          - |         0.00 |
|                   |          |               |          |            |              |
| SelectIndex       | .NET 8.0 |   11.1235 ns |     1.00 |       72 B |         1.00 |
| SelectIndex       | .NET 9.0 |    0.5603 ns |     0.05 |          - |         0.00 |
|                   |          |               |          |            |              |
| SelectMany        | .NET 8.0 |   10.7537 ns |     1.00 |       64 B |         1.00 |
| SelectMany        | .NET 9.0 |    0.9906 ns |     0.09 |          - |         0.00 |
|                   |          |               |          |            |              |
```

## LINQ

```
| Method           | Runtime  | Mean         | Ratio   | Allocated  | Alloc Ratio |
|----------------- |--------- |-------------:|--------:|-----------:|------------:|
| SkipWhile        | .NET 8.0 | 11.2900 ns   |    1.00 |       72 B |        1.00 |
| SkipWhile        | .NET 9.0 |  1.0988 ns   |    0.10 |          - |        0.00 |
|                  |          |              |         |            |             |
| TakeWhile        | .NET 8.0 | 11.8818 ns   |    1.00 |       72 B |        1.00 |
| TakeWhile        | .NET 9.0 |  1.0381 ns   |    0.09 |          - |        0.00 |
|                  |          |              |         |            |             |
| WhereIndex       | .NET 8.0 | 11.1751 ns   |    1.00 |       80 B |        1.00 |
| WhereIndex       | .NET 9.0 |  1.2185 ns   |    0.11 |          - |        0.00 |
```

# .NET JIT – over and over again …

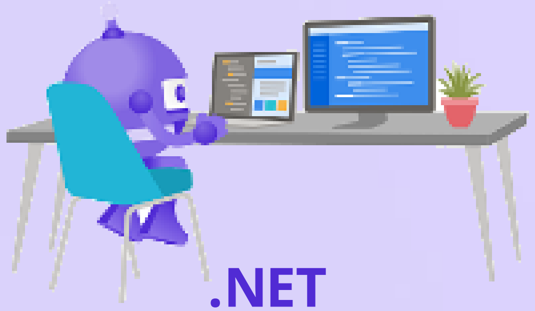
- JIT
- PGO
- Tier 0
- Loops
- Bounds Checks
- Arm64
- ARM SVE
- AVX10.1
- AVX512
- Vectorization
- Branching
- Write Barriers
- Object Stack Allocation
- Inlining
- GC
- VM
- Mono
- Native AOT
- Threading
- Reflection
- Numerics
- Primitive Types
- BigInteger
- TensorPrimitives
- Strings, Arrays, Spans
- IndexOf
- Regex
- Encoding
- Span, Span, and more Span
- Collections
- LINQ
- Core Collections
- Compression
- Cryptography
- Networking
- JSON
- Diagnostics
- Peanut Butter

## Performance Improvement in .NET from Stephen Toub

- **.NET 9** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-9
- **.NET 8** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8
- **.NET 7** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-7
- **.NET 6** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6
- **.NET 5** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5
- **.NET 3.0** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-core-3-0
- **.NET 2.1** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-core-2-1
- **.NET 2.0** https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-core

## Q&A

# Thanks !



**.NET**
your platform for building anything

## About me…

**Mirco Vanini**
Microsoft MVP Developer Technologies

Consultant focused on industrial and embedded solutions using .NET and other native SDKs with over 35 years of experience, XeDotNet community co-founder, speaker and Microsoft MVP since 2012.

MVP Microsoft® Most Valuable Professional

sessionize MOST ACTIVE SPEAKER 2023

sessionize MOST ACTIVE SPEAKER 2024

@MircoVanini
@mircovanini.bsky.social
www.proxsoft.it
https://www.linkedin.com/in/proxsoft
https://sessionize.com/mirco-vanini

## BONUS

## Harness the power of SIMD instructions with Vector

```
private void Normalize(float[] data)
{
    for (int i = 0; i < data.Length; i++)
    {
        data[i] = data[i] / 2f;
    }
}

private void NormalizeWithSIMD(float[] data)
{
    Vector<float> factor = new Vector<float>(0.5f);
    for (int i = 0; i < data.Length; i += Vector<float>.Count)
    {
        Vector<float> vector = new Vector<float>(data, i);
        (vector * factor).CopyTo(data, i);
    }
}
```

```
| Method            | Mean      | Ratio | Rank | Allocated | Alloc Ratio |
|------------------ |----------:|------:|-----:|----------:|------------:|
| Normalize         | 8.089 ms  |  1.00 |    2 |       7 B |        1.00 |
| NormalizeWithSIMD | 3.435 ms  |  0.43 |    1 |       4 B |        0.57 |
```