# .NET Performance

Michael Ciceri

@Laxxifer

michael-ciceri

Owner Laxxifer S.r.l.
IT Consultant

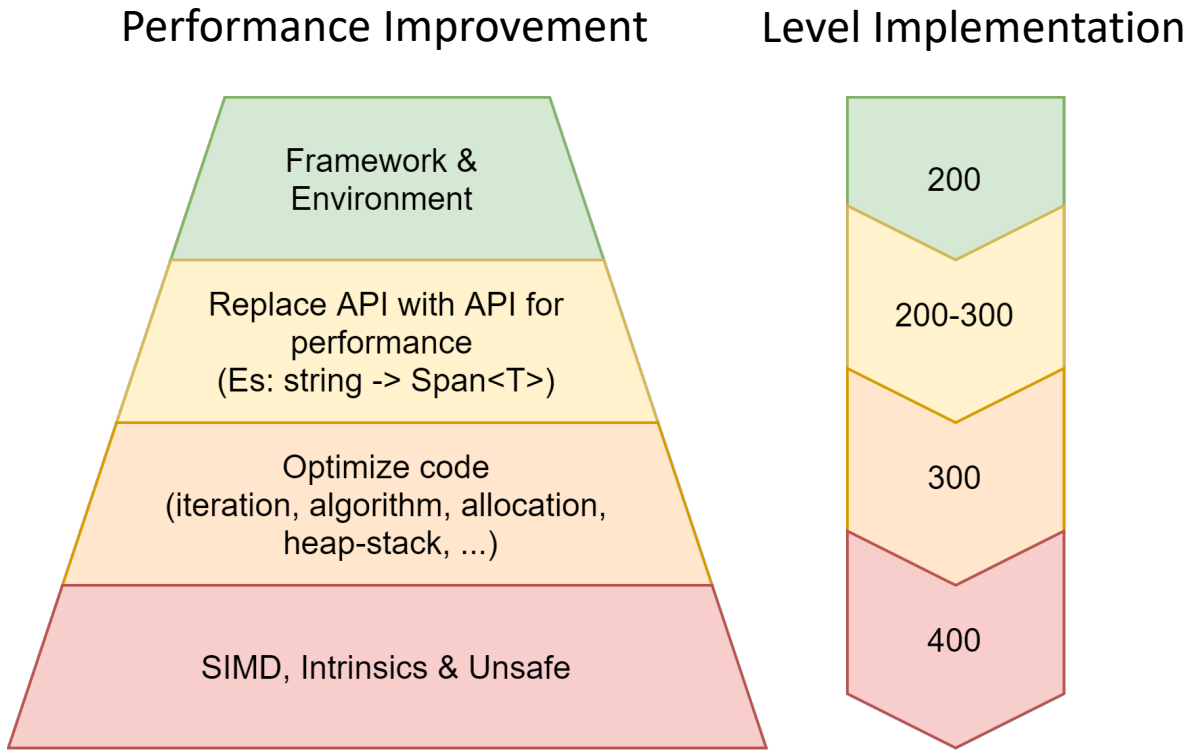

CommunityDays.it

# Ha senso parlare di performance?

- Come Requisito: Ottimizzare i costi e raggiungere un obiettivo.
- Mantenere il controllo e governare il progetto (metriche vs opinioni).
- Comprehension, comprehension, comprehension …

# Agire sempre in ottica di performance?

# .NET Performance



# .NET (Core) Performance

**Compilazione**

- Tiered-Compilation
- ReadyToRun
- Ahead of Time

**API**

- Span<T> / Memory <T>
- Pipeline
- Channel

**Advanced API & Internal**

- nint & nunit
- System.Half
- SkipLocalInit
- Intrinsics & SIMD

**Tools**

dotnet CLI

BenchmarkDotNet

# .NET (Core) Performance

**Compilazione**

- Tiered-Compilation
- ReadyToRun
- Ahead of Time

**API**

- Span<T> / Memory <T>
- Pipeline
- Channel

**Advanced API & Internal**

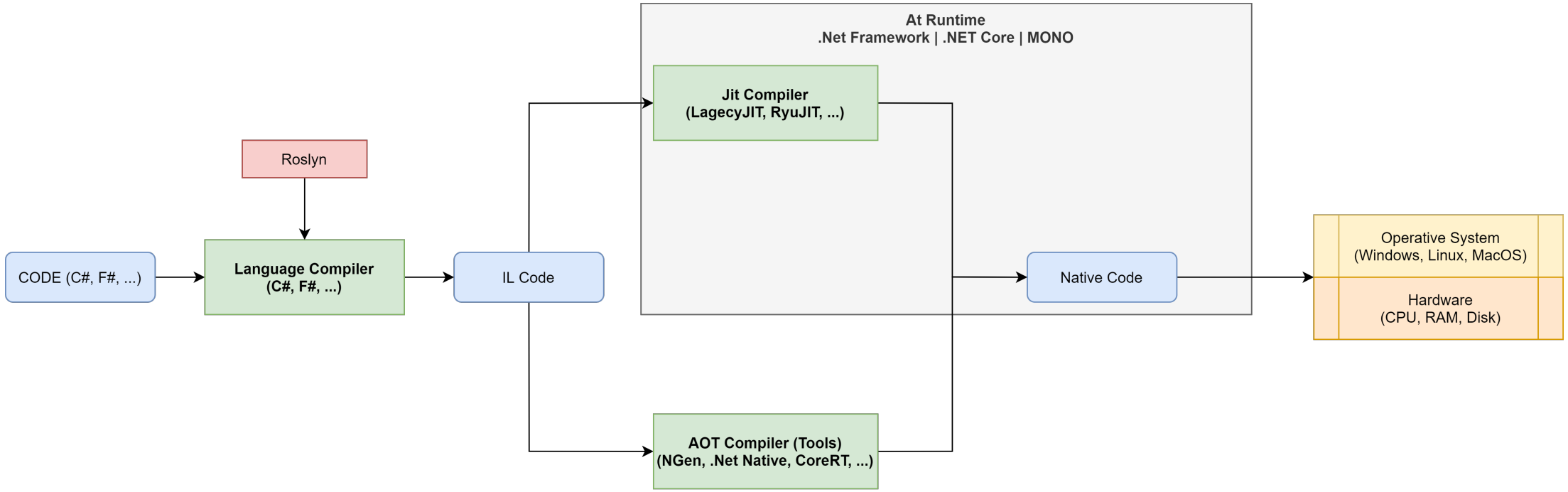
- nint & nunit
- System.Half
- SkipLocalInit
- Intrinsics & SIMD

**Tools**

dotnet CLI

BenchmarkDotNet

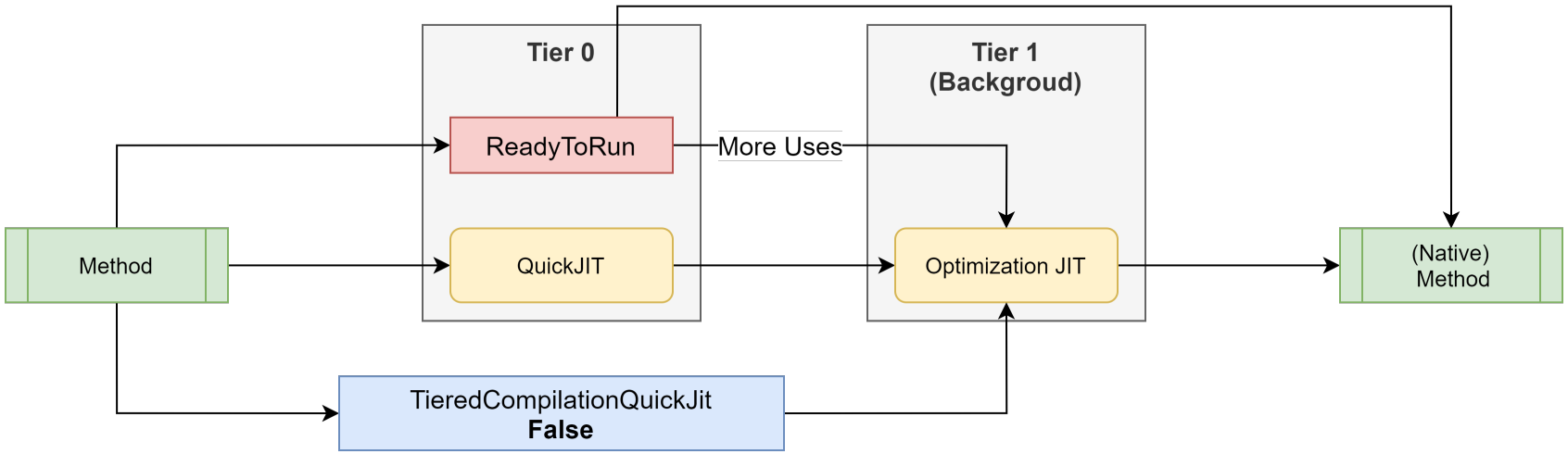
# Compilazione



.NET Core 2.1

# tiered-compilation

- Tramite il runtime, questa **Compilazione a livelli** permette di adattare il compilatore JIT per ottenere prestazioni migliori
- La compilazione JIT avviene su 2 livelli (Tier 0-1)



# ReadyToRun

- Tramite **ReadyToRun (R2R)** è possibile ridurre il tempo di avvio delle applicazioni .NET

```
<PropertyGroup>
  <PublishReadyToRun>true</PublishReadyToRun>
</PropertyGroup>
```

- La compilazione R2R è AOT (Ahead of Time), ma contiene anche il codice intermedio
- È necessario definire in fase di compilazione il RID (es: win-x64)
- Uno svantaggio da considerare è la dimensione degli assembly (2x-3x)

# Compilazione Nativa “e” AOT

Ci sono diversi tools per compilare immagini native:

- **Ngen.exe** compila nativo per .Net Framework
- **CrossGen.exe** compila nativo per .NET Core (CrossGen2 .NET 6.0)

Ci sono diverse toolchain di compilazione AOT:

- **Core RT** (ora NativeAOT) compilazione AOT per applicazioni .NET (single-file o static-dynamic assembly)
- **.NET Native (CoreRT + UTC)** utilizzata per app UWP

# .NET (Core) Performance

**Compilazione**

- Tiered-Compilation
- ReadyToRun
- Ahead of Time

**API**

- Span<T> / Memory <T>
- Pipeline
- Channel

**Advanced API & Internal**

- nint & nunit
- System.Half
- SkipLocalInit
- Intrinsics & SIMD

**Tools**

dotnet CLI

BenchmarkDotNet

.NET Core 2.1

# Span / Memory

*MSDN: Span e Memory sono tipi che rappresentano un'area contigua fortemente tipizzata della memoria arbitraria.*

Per differenziare…

- Span<T> è un tipo **ref Struct** (C# 7.2) che rappresenta un insieme di elementi T attraverso un'implementazione molto leggera **allocato nello stack.**
- Memory<T> è un tipo **Struct** che rappresenta un insieme di elementi T attraverso un'implementazione leggera **allocabile nell'heap gestito**.

.NET Core 2.1

# Span<T> (ref Struct)

**Vantaggi**

- Allocazione e deallocazione più veloce
- É possible puntare a:
  - Memoria gestita
  - Memoria gestita nello stack
  - Memoria nativa

```
Span<byte> spanManage = new byte[1_000];
```

```
Span<byte> spanStack = stackalloc byte[1_000]
```

```
unsafe
{
    var native = Marshal.AllocHGlobal(1_000);
    var spanNative = new Span<byte>(native.ToPointer(), 1_000);
}
```

- Non ci sono operazioni di copia della memoria durante lo slicing!

.NET Core 2.1

# Span<T> (ref Struct)

**Svantaggi**

- Span non supporta il boxing (stack only)
- Non può essere utilizzato in operazioni asincrone (await async)
- Non può essere usato come field in un tipo riferimento

Allocation | Time-Execution

.NET Core 2.1

# Span

```
public List<String> _Data = new() {"A34345-213", "Bwr-2133", "C234-345", "Dwer-456", "E234-678", "Fwr-2342", "G32-2342"};
```

```
public void SumLengthString()
{
    int cont = 0;

    for (int i = 0; i < this.N; i++)
        cont += this._Data.Sum(letter => SplitString(letter));

    static int SplitString(string l)
    {
        var index = l.IndexOf('-');
        return l.Substring(index, l.Length - index).Length;
    }
}
```

```
public void SumLengthSpan()
{
    int cont = 0;

    for (int i = 0; i < this.N; i++)
        cont += this._Data.Sum(letter => SplitSpan(letter.AsSpan()));

    static int SplitSpan(ReadOnlySpan<char> l)
    {
        var index = l.IndexOf('-');
        return l.Slice(index, l.Length - index).Length;
    }
}
```

| Method | N | Mean | Ratio | RatioSD | Gen 0 | Gen 1 | Gen 2 | Allocated |
|---|---|---|---|---|---|---|---|---|
| SumLengthString | 100000 | 28.75 ms | 1.00 | 0.00 | 3000.0000 | - | - | 25.18 MB |
| SumLengthSpan | 100000 | 18.84 ms | 0.65 | 0.07 | - | - | - | 3.81 MB |

- Anche se l'esempio è fine a se stesso 😔 è evidente la differenza di memoria allocata e dei tempi di esecuzione

.NET Core 2.1

# E Memory…

- Memory<T> punta solo alla memoria gestita, di conseguenza può essere utilizzato negli ambiti in cui Span non poteva essere utilizzato
- Espone Span come proprietà e quindi potrebbe essere «immaginato come un wrapper» dello stesso
- Oltre a supportare in fase di creazione array e stringhe può essere istanziato anche da **IMemoryOwned**

```
IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();

//OR..

using IMemoryOwner<char> ownerUsing = MemoryPool<char>.Shared.Rent();
var memory = owner.Memory;
var span = memory.Span;
```

Allocation

.NET Standard

# Pipelines

- System.IO.Pipelines è una libreria che permette di eseguire operazioni I/O ad alte prestazioni

```
var pipe = new Pipe();
var reader = pipe.Reader;
var writer = pipe.Writer;

writer.WriteAsync(new ReadOnlyMemory<byte>(new byte[10]), CancellationToken.None);

ReadOnlySequence<byte> result = await reader.ReadAsync(CancellationToken.None).Buffer;
```

Bad code!

**Vantaggi**

- Riduce le allocazioni sulle scritture e letture
- È possibile gestire i tempi di «Pause» e «Resume» sulla scrittura in modo da poter ottimizzare le attese tra letture e scritture

Allocation

.NET Standard

# Channel

- System.Threading.Channel può essere vista come una struttura dati a cui un consumatore può accedere in sicurezza grazie ad un sincronizzatore

```
var channel = Channel.CreateUnbounded<int>();
var reader = channel.Reader;
var write = channel.Writer;

var str = await reader.ReadAsync(CancellationToken.None);
await write.WriteAsync(1);
```

**Vantaggi**

- Riduce le allocazioni tramite l'uso di ValueTask sfruttando le interfacce anche quando il codice è eseguito in contesto asincrono
- L'implementazione nativa in .NET (Core) è più ottimizzata rispetto a quella via nuget per .Net Framework

# .NET (Core) Performance

**Compilazione**

- Tiered-Compilation
- ReadyToRun
- Ahead of Time

**API**

- Span<T> / Memory <T>
- Pipeline
- Channel

**Advanced API & Internal**

- nint & nunit
- System.Half
- SkipLocalInit
- Intrinsics & SIMD

**Tools**

dotnet CLI

BenchmarkDotNet

Time-Execution | Allocation

.NET 5.0

# nint & nuint

- A partire da C# 9 sono state rese disponibili queste 2 parole chiave che rappresentano **valori interi a dimensione nativa**

# System.Half

- System.Half rappresenta un numero a virgola mobile da 16bit

Time-Execution

.NET 5.0

# SkipLocalInit

- Tramite attributo SkipLocalInit è possibile istruire il compilatore di non inizializzare le “variabili locali” al valore di default
- É possibile sfruttarlo anche in singole variabili locali o porzioni di struct tramite

```
Unsafe.SkipInit(out i)
```

```
[Params(10, 1_000, 10_000, 100_000, 1_000_000)]
public int Size { get; set; }

[Benchmark(Baseline = true)]
public byte InitLocals()
{
    Span<byte> s = stackalloc byte[Size];
    return s[0];
}

[Benchmark]
[SkipLocalsInit]
public byte SkipInitLocals()
{
    Span<byte> s = stackalloc byte[Size];
    return s[0];
}
```

| Method | Size | Mean | Error | Median | Ratio |
|---|---|---|---|---|---|
| InitLocals | 10 | 2.787 us | 8.699 us | 0.2000 us | 1.00 |
| SkipInitLocals | 10 | 2.677 us | 8.233 us | 0.2000 us | 1.22 |
| | | | | | |
| InitLocals | 1000 | 2.648 us | 8.244 us | 0.2000 us | 1.00 |
| SkipInitLocals | 1000 | 2.700 us | 8.397 us | 0.2000 us | 1.17 |
| | | | | | |
| InitLocals | 10000 | 3.276 us | 8.700 us | 0.5000 us | 1.00 |
| SkipInitLocals | 10000 | 2.826 us | 8.769 us | 0.2000 us | 0.37 |
| | | | | | |
| InitLocals | 100000 | 7.098 us | 10.115 us | 3.7000 us | 1.00 |
| SkipInitLocals | 100000 | 3.136 us | 9.543 us | 0.3000 us | 0.09 |
| | | | | | |
| InitLocals | 1000000 | 81.892 us | 27.432 us | 38.6000 us | 1.00 |
| SkipInitLocals | 1000000 | 7.889 us | 21.693 us | 1.4000 us | 0.04 |

Unsafe

Unsafe

CPU-Execution

.NET Core 3.0

# SIMD & Intrinsics

- Tramite le API System.Runtime.Intrinsics è ora possibile sfruttare la parellizzazione a livello di CPU tramite implentazione SIMD specifica!

```
using System.Runtime.Intrinsics.X86; //esiste specifico per Arm
var vresult = Vector128<int>.Zero;

if (Ssse3.IsSupported)
    vresult = Ssse3.HorizontalAdd(vresult, vresult);

//…
if (Sse2.IsSupported)
    vresult = Sse2.Add(vresult, vresult);
```

**Vantaggi:** Performance!

**Svantaggi**

- Complessità e verbosità del codice (CPU e target platform specific)
- Implementare a mano una fallback software
- Comprendere e conoscere le istruzioni dei processori

`Avx2.X64.IsSupported`

Level 400

Level 400

# Guidelines

- Security e Performance spesso collidono con effetti molto gravi in entrambe le direzioni
- Performance is a feature… Ma solo davanti a metriche reali e replicabili
- Environment e tanto importante quanto il codice

# Grazie!

- Il materiale sarà online nei prossimi giorni su http://www.communitydays.it