



Performance Improvements in .NET 8



Mirco Vanini
@MircoVanini

xdotnet.org



Mirco Vanini

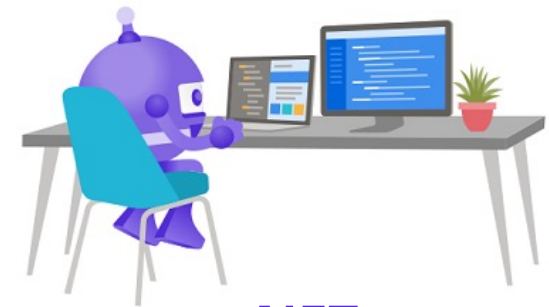
Microsoft MVP Developer Technologies

Consulente con oltre 30 anni di esperienza, specializzato in soluzioni industriali ed embedded, cofondatore della community XeDotNet, relatore e Microsoft MVP dal 2012



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

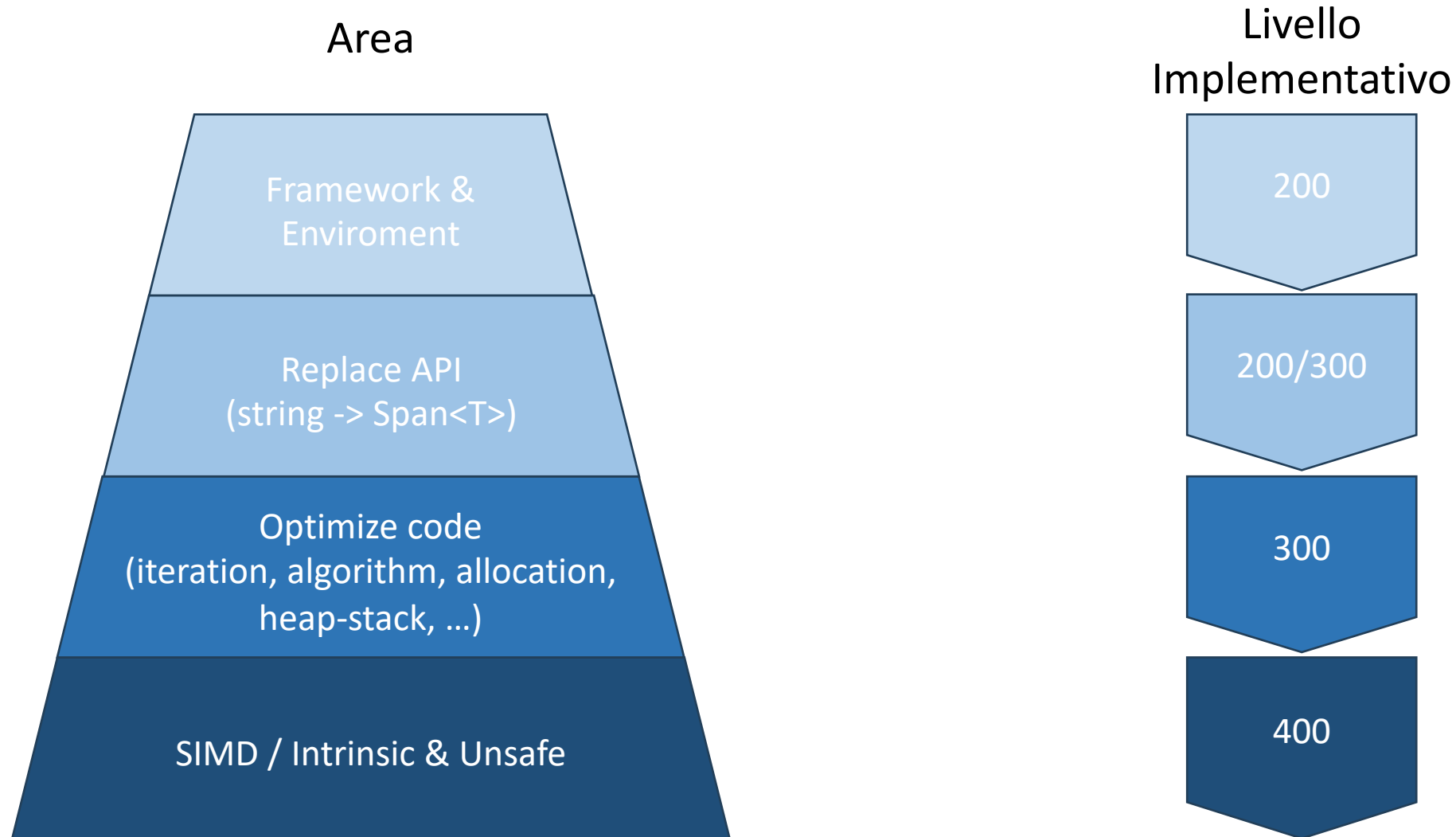
Microsoft® MVP Developer Technologies



.NET
your platform for building anything



- Introduzione
- JIT – On Stack Replacement
- JIT – PGO
- Reflection
- String
- Regular Expression
- Native AOT
- JSON
- ...





.NET 8 – sorpresa ! è più veloce di .NET 7 il quale era più veloce di .NET 6, il quale era più veloce di .NET5 che era più veloce di...

Molte delle funzionalità prestazionali non sono specificatamente guidate dallo sviluppatore, ma sono invece integrate nel FW

Da quando .NET Core è entrato in scena più di nove anni fa, le prestazioni sono state parte integrante della cultura di .NET.

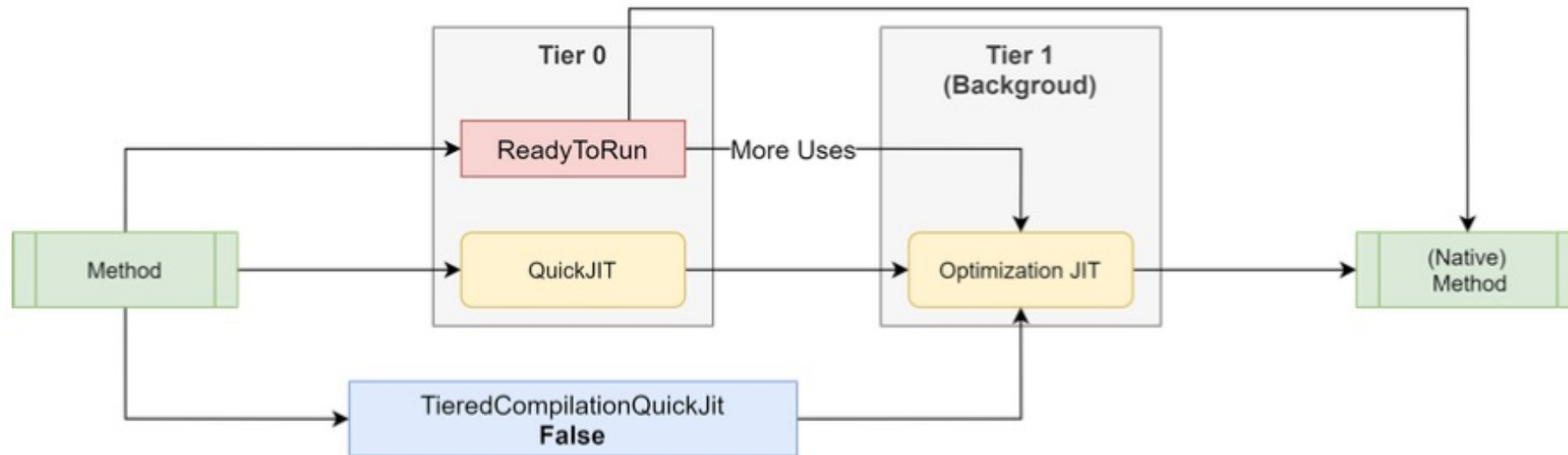
JIT – Compilazione a più livelli (Tiered Compilation)



La compilazione JIT è fantastica... ma presenta anche degli svantaggi, come una velocità di avvio più lenta

La compilazione a livelli è un ottimo compromesso tra JIT e codice nativo, è stata introdotta in .NET Core 3.0 e da allora è stata costantemente migliorata

Consenti un avvio più rapido senza sacrificare la qualità del codice Ma disattivato per impostazione predefinita per i metodi con cicli...



JIT – Sostituzione sullo stack (On-Stack Replacement)



In .NET 8 anche i metodi con cicli traggono vantaggio dalla compilazione a più livelli. Ciò si ottiene tramite la sostituzione sullo stack (OSR).

L'OSR fa sì che il JIT non solo consideri la compilazione iniziale per il numero di invocazioni, ma anche i cicli per il numero di iterazioni elaborate.

Quando il numero di iterazioni supera un limite predeterminato, proprio come con il conteggio delle invocazioni, il JIT compila una nuova versione ottimizzata del metodo

DemoOSR / DemoOSRStatic

JIT – Sostituzione sullo stack (On-Stack Replacement)



```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.2506/22H2/2022Update/SunValley2)
Intel Core i9-9880H CPU 2.30GHz, 1 CPU, 8 logical and 8 physical cores
.NET SDK=8.0.100-rc.2.23502.2
```

```
[Host]      : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2
Job-JUHWBR  : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2
Job-UOWGAO  : .NET 7.0.13 (7.0.1323.51816), X64 RyuJIT AVX2
Job-OAOHNN  : .NET 8.0.0 (8.0.23.47906), X64 RyuJIT AVX2
```

Method	Runtime	Mean	Ratio	Code Size
Compute	.NET 6.0	926.4 us	1.00	66 B
Compute	.NET 7.0	243.1 us	0.26	17 B
Compute	.NET 8.0	242.8 us	0.26	17 B

JIT – Sostituzione sullo stack (On-Stack Replacement)



.NET 6.0.24 - X64 RyuJIT AVX2

```
Program.Compute()
    push    rdi
    push    rsi
    sub     rsp,28
    xor     esi,esi
    xor     edi,edi
    mov     rcx,7FF95AED2D08
    mov     edx,5
    call    CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    mov     eax,[7FF95AED2D40]
M00_L00:
    add     esi,edi
    cmp     eax,7E5
    jne     short M00_L01
    add     esi,edi
M00_L01:
    inc     edi
    cmp     edi,0F4240
    jl     short M00_L00
    mov     eax,esi
    add     rsp,28
    pop     rsi
    pop     rdi
    ret
; Total bytes of code 66
```

.NET 7.0.13 / .NET 8.0.0 - X64 RyuJIT AVX2

```
; Program.Compute()
    xor     eax,eax
    xor     edx,edx
M00_L00:
    add     eax,edx
    inc     edx
    cmp     edx,0F4240
    jl     short M00_L00
    ret
; Total bytes of code 17
```



L'ottimizzazione guidata dal profilo (PGO) esiste da decenni, per molti linguaggi e ambienti, incluso il mondo .NET.

Il flusso tipico prevede che si crei l'applicazione con qualche strumentazione aggiuntiva, quindi si esegua l'applicazione su scenari chiave, si raccolgano i risultati di quella strumentazione e quindi si ricostruisca l'applicazione, inserendo i dati di strumentazione nell'ottimizzatore, consentendogli di utilizzare la conoscenza di come viene eseguito il codice per influire sulla sua ottimizzazione

Questo approccio viene definito “PGO statico”.



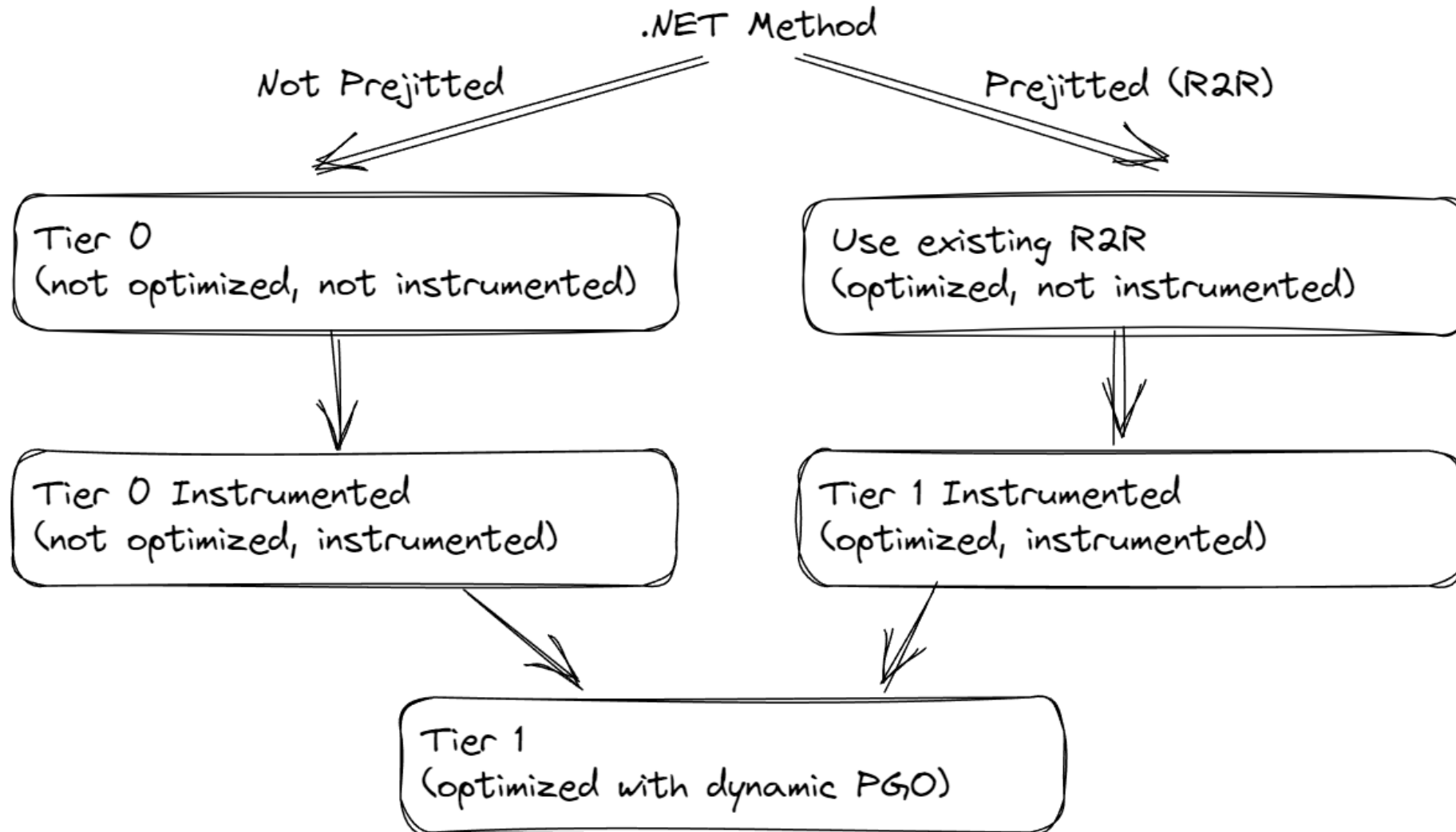
"Dynamic PGO" è simile, tranne per il fatto che non è richiesto alcuno sforzo su come viene creata l'applicazione, sugli scenari su cui viene eseguita o altro.

Presentato per la prima volta in anteprima in .NET 6, disattivato per impostazione predefinita anche in .NET 7, ora in .NET 8 è attivo per default.

È stata modificata la compilazione a livelli aggiungendo più livelli, anche se continuiamo a riferirci a quello non ottimizzato come "livello 0" e a quello ottimizzato come "livello 1".

La strumentazione non è gratuita, l'obiettivo del livello 0 è rendere la compilazione il più economica possibile

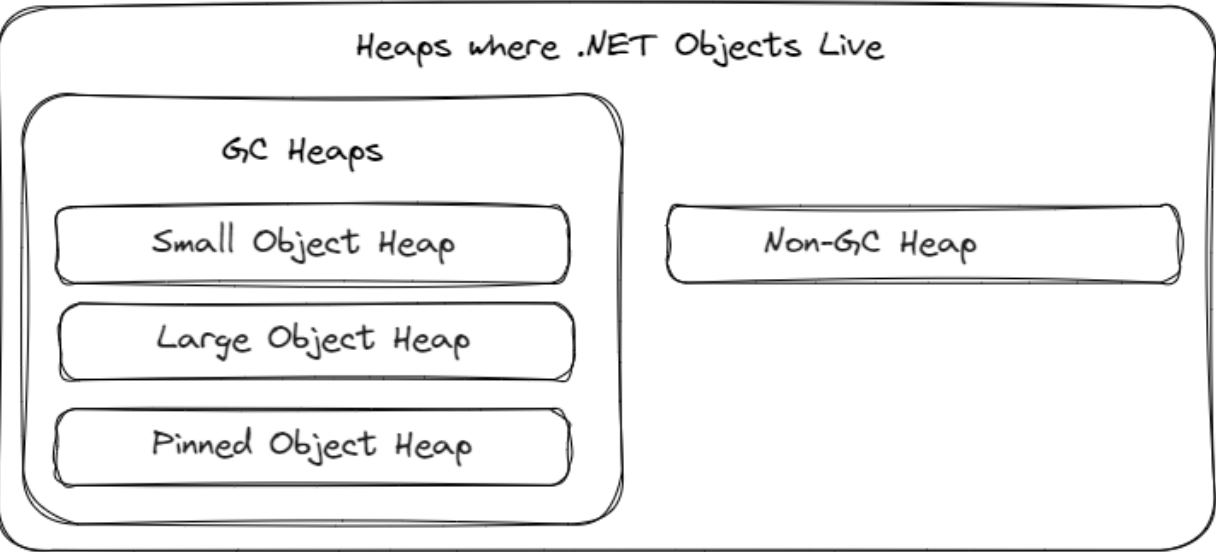
JIT – PGO (Profile Guided Optimization)



Non-GC Heap - "Frozen Segments"



.NET 8 introduce un nuovo meccanismo utilizzato dal JIT il Non-GC Heap (un'evoluzione del vecchio concetto di "Frozen Segments" utilizzato da Native AOT)



```
.NET 6/7
; Tests.GetPrefix()
mov rax,126A7C01498
mov rax,[rax]
ret
; Total bytes of code 14
```

```
.NET 8
; Tests.GetPrefix()
mov rax,227814EAEA8
ret
; Total bytes of code 11
```

DemoPGO

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
GetTestsType	.NET 6.0	1.2022 ns	1.000
GetTestsType	.NET 7.0	0.2469 ns	0.206
GetTestsType	.NET 8.0	0.0024 ns	0.002

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
GetPrefix	.NET 6.0	0.8072 ns	1.00
GetPrefix	.NET 7.0	0.6577 ns	0.81
GetPrefix	.NET 8.0	0.1185 ns	0.14

(GDV) - Devirtualizzazione protetta



Una delle principali ottimizzazioni dei feed PGO dinamici è la capacità di devirtualizzare le chiamate virtuali e di interfaccia per sito di chiamata. Come notato, il JIT tiene traccia dei tipi concreti utilizzati e quindi può generare un percorso rapido per il tipo più comune; questo è noto come devirtualizzazione protetta (GDV).

```
int result = _valueProducer!.GetType() == typeof(Producer42) ?
    Unsafe.As<Producer42>(_valueProducer).GetValue() :
    _valueProducer.GetValue();
return result * _factor;
```

```
int result = _valueProducer!.GetType() == typeof(Producer42) ?
    42 :
    _valueProducer.GetValue();
return result * _factor
```

DemoPGO

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
GetValue	.NET 6.0	2.0347 ns	1.00
GetValue	.NET 7.0	1.6867 ns	0.79
GetValue	.NET 8.0	0.2807 ns	0.13



[Tiering and Dynamic PGO](#)

[Vectorization](#)

[Branching](#)

[Bounds Checking](#)

[Constant Folding](#)

[Non-GC Heap](#)

[Zeroing](#)

[Value Types](#)

[Casting](#)

[Peephole Optimizations](#)



L'invocazione della Reflection ha storicamente avuto spese generali non banali
Gli sviluppatori hanno risolto il problema con Reflection emit

Approccio con Reflection emit è ora integrato!

```
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	Ratio
InvokeZero	.NET 6.0	50.180 ns	1.00
InvokeZero	.NET 7.0	10.896 ns	0.22
InvokeZero	.NET 8.0	7.324 ns	0.15
InvokeOne	.NET 6.0	98.758 ns	1.00
InvokeOne	.NET 7.0	28.832 ns	0.31
InvokeOne	.NET 8.0	20.824 ns	0.21


```
// 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	2,114.69 µs	1.00
Count	.NET 7.0	Sherlock	617.44 µs	0.29
Count	.NET 8.0	Sherlock	168.73 µs	0.09
Count	.NET 6.0	elementary	2,435.47 µs	1.00
Count	.NET 7.0	elementary	664.00 µs	0.27
Count	.NET 8.0	elementary	65.57 µs	0.03



```
// 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 CountComparison()
{
    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	Ratio
-----	-----	-----	-----	-----	-----
CountComparison	.NET 6.0	Ordinal	elementary	1,138.42 μs	1.00
CountComparison	.NET 7.0	Ordinal	elementary	142.52 μs	0.13
CountComparison	.NET 8.0	Ordinal	elementary	140.57 μs	0.12
CountComparison	.NET 6.0	OrdinalIgnoreCase	elementary	2,447.14 μs	1.00
CountComparison	.NET 7.0	OrdinalIgnoreCase	elementary	687.40 μs	0.28
CountComparison	.NET 8.0	OrdinalIgnoreCase	elementary	66.23 μs	0.03

```
private byte[] _data = new byte[98];
```

```
[Benchmark]
```

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

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
Contains	.NET 6.0	17.283 ns	1.00
Contains	.NET 7.0	5.441 ns	0.27
Contains	.NET 8.0	4.470 ns	0.26

```
private int[] _dataInt = new int[10240];
```

```
[Benchmark]
```

```
public int IndexOf() => _dataInt.AsSpan().IndexOf(42);
```

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
IndexOf	.NET 6.0	3.107 µs	1.00
IndexOf	.NET 7.0	1.046 µs	0.34
IndexOf	.NET 8.0	1.024 µs	0.33

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

```
[Benchmark]
```

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

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
Replace	.NET 6.0	1,635.37 ns	1.00
Replace	.NET 7.0	75.04 ns	0.05
Replace	.NET 8.0	63.59 ns	0.04

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

Method	Runtime	text	Mean
-----	-----	-----	-----:
StartsWith	.NET 4.8	http://microsoft.com	21.590 ns
OpenCoded	.NET 4.8	http://microsoft.com	2.048 ns

```
[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] == '/';
```

Method	Runtime	text	Mean
-----	-----	-----	-----:
StartsWith	.NET 6.0	http://microsoft.com	7.2537 ns
OpenCoded	.NET 6.0	http://microsoft.com	1.369 ns

Method	Runtime	text	Mean
-----	-----	-----	-----:
StartsWith	.NET 7.0	http://microsoft.com	0.6405 ns
OpenCoded	.NET 7.0	http://microsoft.com	1.299 ns

Method	Runtime	text	Mean
-----	-----	-----	-----:
StartsWith	.NET 8.0	http://microsoft.com	0.5430 ns
OpenCoded	.NET 8.0	http://microsoft.com	1.270 ns

Lezione: le soluzioni alternative dovrebbero essere riviste!

Espressioni Regolari



```
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;
}
```

Non-Prefix String Search

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
SubstringSearch	.NET 4.8	3,625.875 µs	1.000
SubstringSearch	.NET 6.0	877.410 µs	0.269
SubstringSearch	.NET 7.0	12.934 µs	0.003
SubstringSearch	.NET 8.0	9.835 µs	0.002

Espressioni Regolari



```
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 4.8	11.019.362 µs	1.0000
Email	.NET 6.0	611.837 µs	0.0480
Email	.NET 7.0	6.571 µs	0.0005
Email	.NET 8.0	4.007 µs	0.0003

Literals after loops

.NET Nativo AOT (Ahead Of Time)



L'AOT nativo è diverso. È un'evoluzione di CoreRT, che a sua volta era un'evoluzione di .NET Native, ed è completamente privo di JIT.

Il file binario che risulta dalla pubblicazione di una build è un eseguibile completamente autonomo nel formato di file specifico della piattaforma di destinazione (ad esempio COFF su Windows, ELF su Linux, Mach-O su macOS) senza dipendenze esterne diverse da quelle standard su quella piattaforma (ad esempio libc).

Ed è del tutto nativo: niente IL, niente JIT, niente di niente. Tutto il codice richiesto viene compilato e/o collegato all'eseguibile, incluso lo stesso GC utilizzato con app e servizi .NET standard e un runtime minimo che fornisce servizi relativi al threading e simili.

[Native AOT deployment overview](#)

.NET Nativo AOT (Ahead Of Time)



Porta anche delle limitazioni: nessun JIT significa nessun caricamento dinamico di assembly arbitrari (ad esempio `Assembly.LoadFile`) e nessuna emissione di riflessioni (ad esempio `DynamicMethod`), tutto ciò che è compilato è collegato all'app, questo significa che più funzionalità sono utilizzate (o potrebbe essere utilizzate) più grande è la tua distribuzione, ecc.

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

Con JIT, `IsDynamicCodeCompiled` è vero ma, con l'AOT nativo, è falso.

.NET Nativo AOT (Ahead Of Time)



ASP.NET Core 8.0 introduce il supporto per [.NET Native ahead-of-time \(AOT\)](#).

In .NET 8, non tutte le funzionalità di ASP.NET Core sono compatibili con AOT nativo. Per un elenco dei problemi AOT nativi noti, vedere questo [link](#)



.NET Nativo AOT (Ahead Of Time)



.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](#)



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

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

I file binari R2R migliorano le prestazioni di avvio riducendo la quantità di lavoro che il compilatore just-in-time (JIT) deve eseguire durante il caricamento dell'applicazione. I file binari contengono codice nativo simile rispetto a quello che produrrebbe il JIT. Tuttavia, i file binari R2R sono più grandi perché contengono sia il codice IL (Intermediate Language), ancora necessario per alcuni scenari, sia la versione nativa dello stesso codice.

L'utilizzo di Composite ReadyToRun è consigliato solo per le applicazioni che disabilitano la compilazione a livelli o per le applicazioni in esecuzione su Linux che cercano il miglior tempo di avvio con una distribuzione autonoma.

Un focus significativo in .NET 8 è stato il miglioramento e la riduzione del codice sorgente autogenerato per la chiamata JsonSerializer.

È stata migliorata la serializzazione polimorfica, il supporto per DateOnly/TimeOnly, il supporto per IEnumerable<T> e JsonDocument nella generazione del codice.

Una delle maggiori insidie in termini di prestazioni che abbiamo visto è relativa al modo in cui la libreria memorizza i dati nella cache. Per ottenere buone prestazioni di serializzazione e deserializzazione quando il generatore di origine non viene utilizzato, System.Text.Json utilizza emit della riflessione per generare codice personalizzato per la lettura/scrittura dei membri dei tipi elaborati

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

public class MyAmazingClass
{
    public int Value { get; set; }
}

[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());
```

Method	Runtime	Mean	Ratio
-----	-----	-----:	-----:
ImplicitOptions	.NET 6.0	178.3 ns	1.00
ImplicitOptions	.NET 7.0	168.2 ns	0.94
ImplicitOptions	.NET 8.0	165.2 ns	0.93
WithCached	.NET 6.0	185.4 ns	1.04
WithCached	.NET 7.0	181.2 ns	1.02
WithCached	.NET 8.0	150.9 ns	0.85
WithoutCached	.NET 6.0	100,414.7 ns	564.44
WithoutCached	.NET 7.0	720.1 ns	4.04
WithoutCached	.NET 8.0	543.0 ns	3.05



[Native AOT](#)

[VM](#)

[GC](#)

[Mono](#)

[Threading](#)

[Reflection](#)

[Exceptions](#)

[Primitives](#)

[Strings, Arrays, and Spans](#)

[Collections](#)

[File I/O](#)

[Networking](#)

[JSON](#)

[Cryptography](#)

[Logging](#)

[Configuration](#)

[Peanut Butter](#)

[Performance Improvements in .NET 8](#)

[Performance Improvements in ASP.NET Core 8](#)

1289 punti, più di 500 PRs/miglioramenti discussi (> 20% dall'esterno del team .NET)

Grazie !

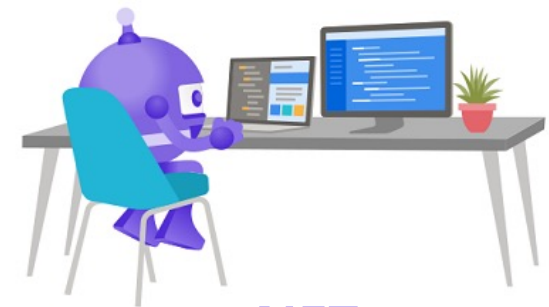


@MircoVanini

www.proxsoft.it

<https://www.linkedin.com/in/proxsoft>

Microsoft® MVP Developer Technologies



.NET

your platform for building anything