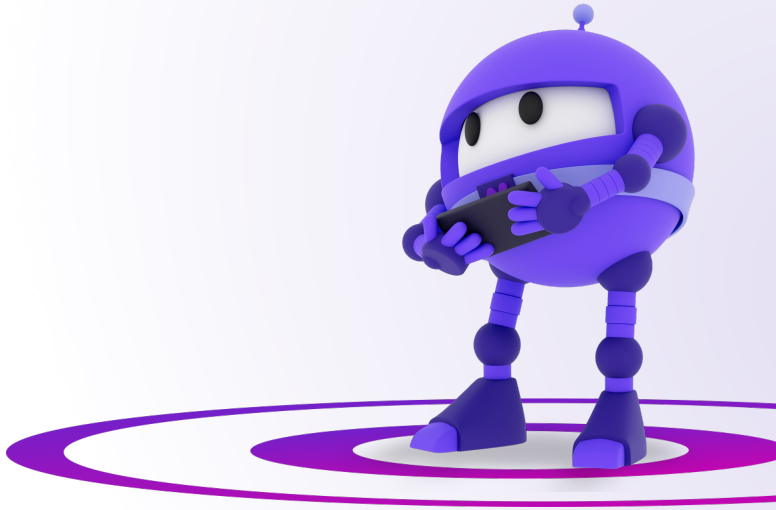
Tabya Conf 2024

Intelligenza Artificiale e Sviluppo di applicazioni moderne

. NET 8 C# Language & Performance Improvement

Mirco Vanini - Marco Parenzan

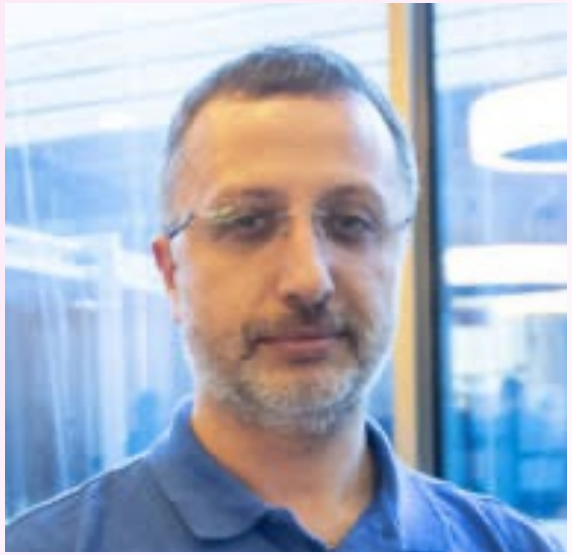


I nostri sponsor

MESA.

OVERNET.

Marco Parenzan



MARCO PARENZAN

beanTech
IT moves your business

Microsoft® Most Valuable Professional

innova community

WPC

marco_parenzan

marcoparenzan

marcoparenzan

marcoparenzan

Mirco Vanini

Consulente con oltre 35 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 - Produttività e Performance

- Produttività: lavorare sullo stile del codice per scriverne meno e meglio
- Performance: conoscere e sfruttare i cambiamenti sottostanti al JIT / BCL

Produttività: alcune novità di .NET 8 e C# 12

Marco Parenzan

Performance: alcuni dei tanti miglioramenti introdotti da .NET 8

Mirco Vanini

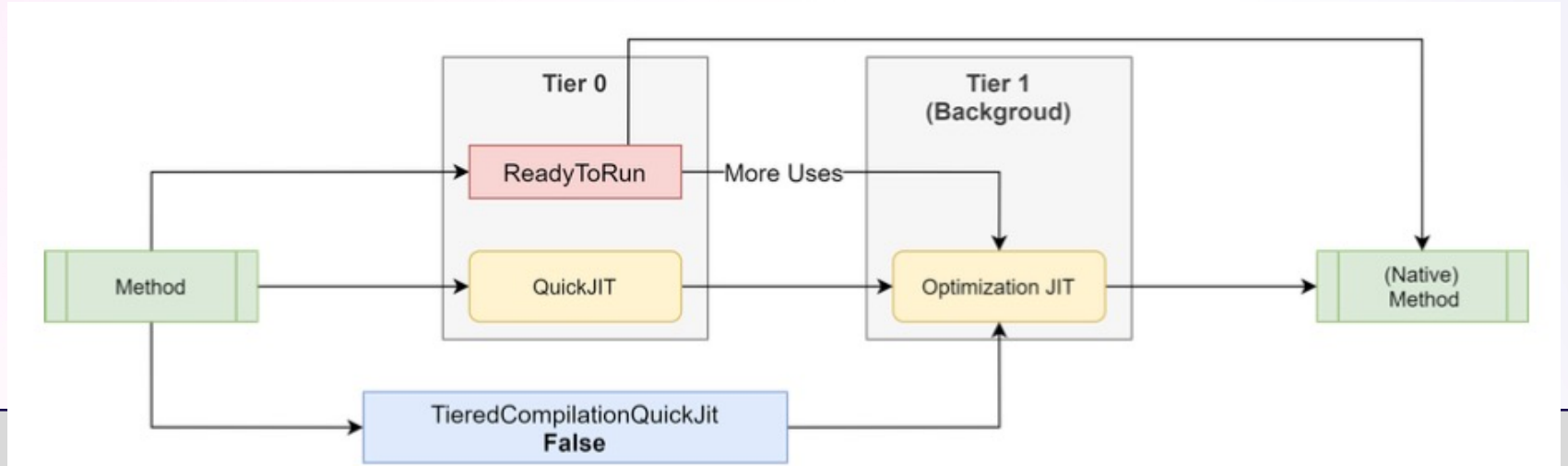
Performance

- .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.

Performance is the feature !

JIT – Compilazione a più livelli

- 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 OSR

- 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 OSR

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.2715/22H2/2022Update/SunValley2)
Intel Core i9-9880H CPU 2.30GHz, 1 CPU, 8 logical and 8 physical cores
.NET SDK=8.0.100
  [Host]     : .NET 6.0.25 (6.0.2523.51912), X64 RyuJIT AVX2
  Job-TOBQYK : .NET 6.0.25 (6.0.2523.51912), X64 RyuJIT AVX2
  Job-VYYFBF : .NET 7.0.14 (7.0.1423.51910), X64 RyuJIT AVX2
  Job-JHPHBL : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
```

Method	Runtime	Mean	Ratio	Code Size
Compute	.NET 6.0	854.1 µs	1.00	66 B
Compute	.NET 7.0	237.0 µs	0.28	17 B
Compute	.NET 8.0	231.6 µs	0.27	17 B

JIT – Sostituzione sullo stack OSR

.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
```

JIT – Ottimizzazione guidata dal profilo PGO

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

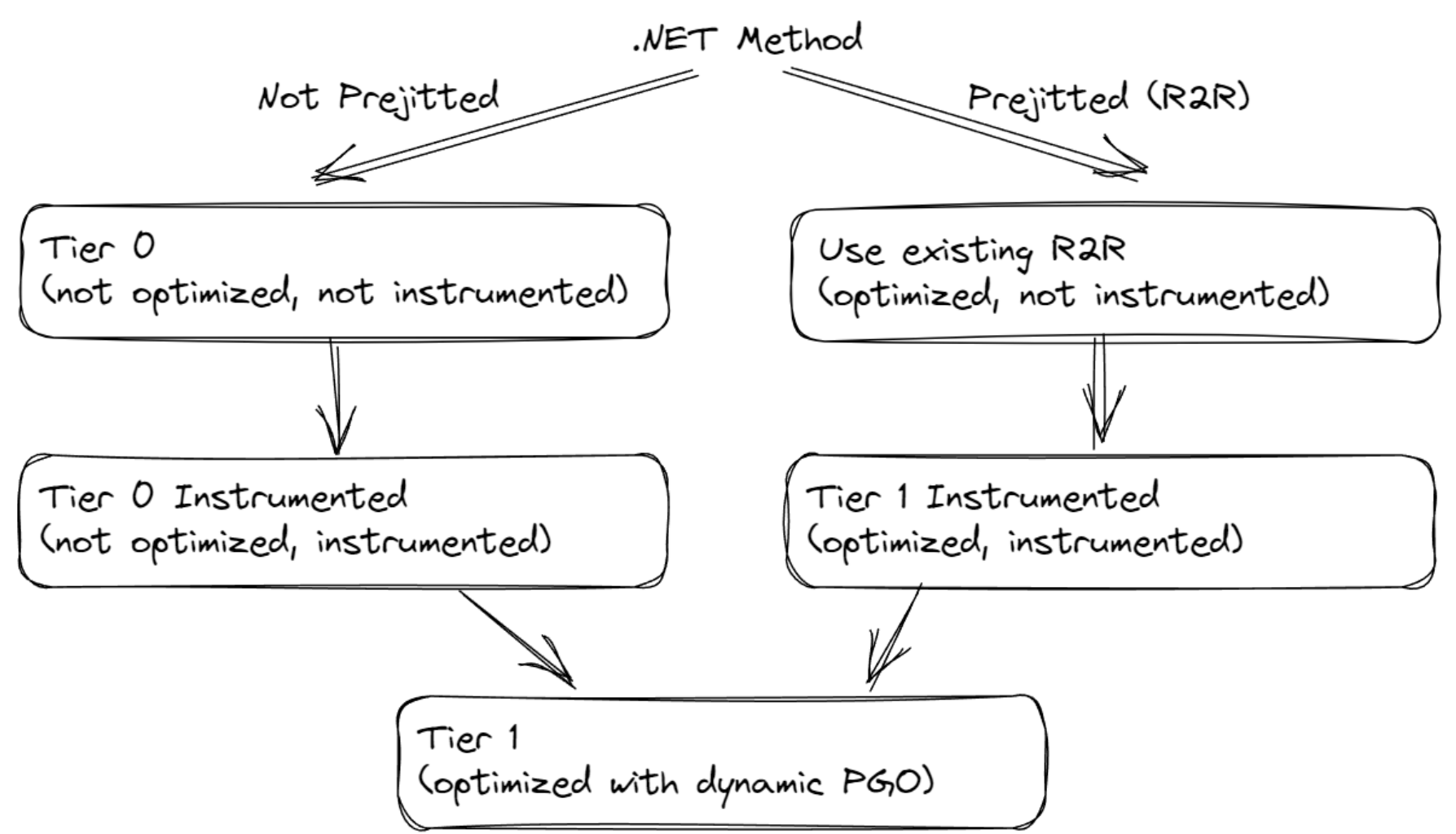
- Il flusso tipico prevede che si crei l'applicazione con qualche strumentazione aggiuntiva, quindi si esegua l'applicazione su scenari chiave, si raccolgano i risultati di quella strumentazione e quindi si ricostruisca l'applicazione, inserendo i dati di strumentazione nell'ottimizzatore, consentendogli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

- Questo approccio viene definito “PGO statico”.

JIT – Ottimizzazione guidata dal profilo PGO

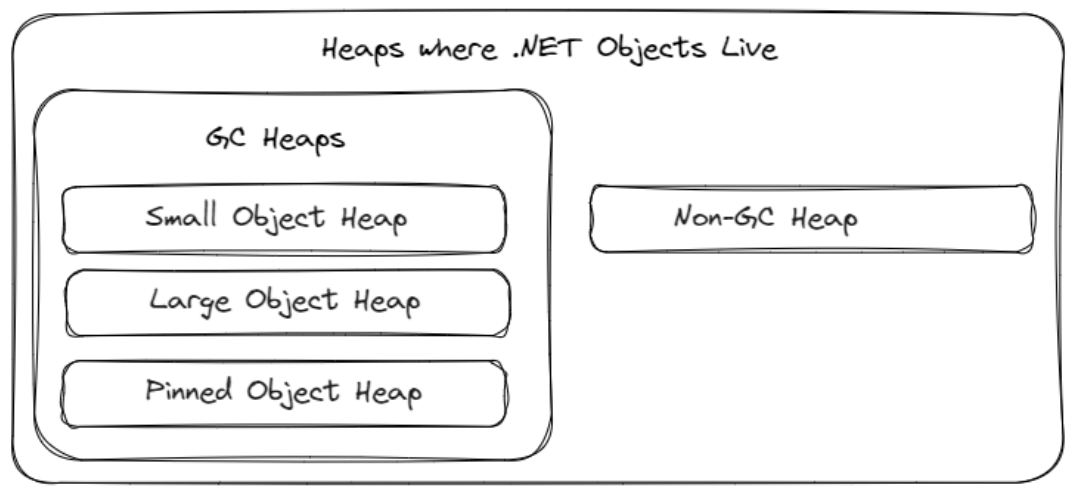
- "Dynamic PGO" è simile, tranne per il fatto che non è richiesto alcuno sforzo su come viene creata l'applicazione, sugli scenari su cui viene eseguita o altro.
- Presentato per la prima volta in anteprima in .NET 6, disattivato per impostazione predefinita anche in .NET 7, ora in .NET 8 è attivo per 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 – Ottimizzazione guidata dal profilo PGO



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

Devirtualizzazione protetta DGV

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;
```

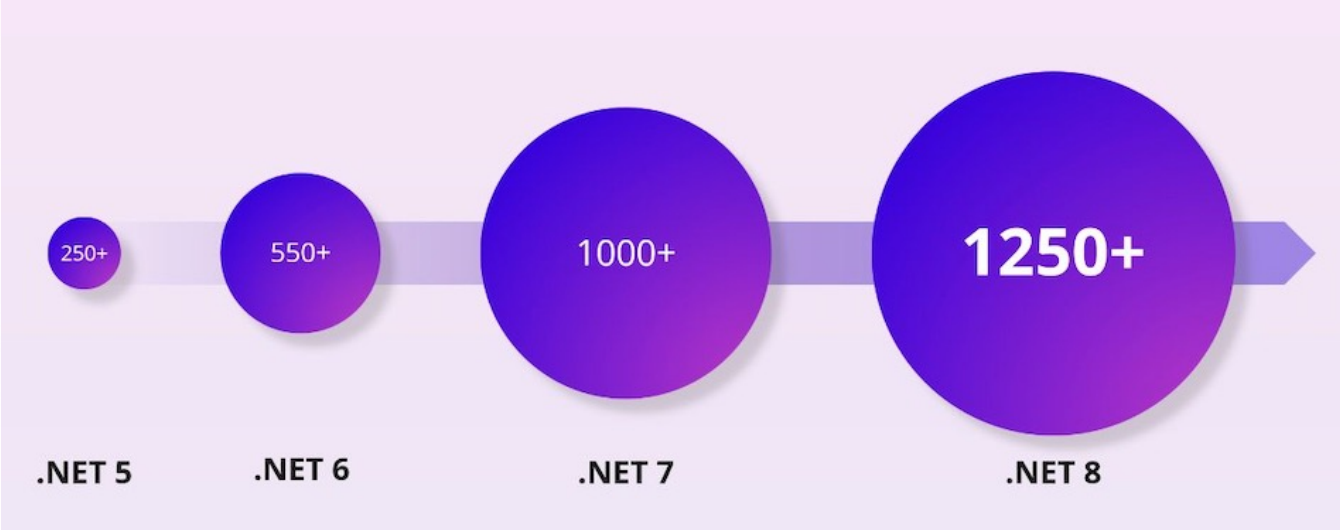
```
int result = _valueProducer!.GetType() == typeof(Producer42) ?
                42 :
                _valueProducer.GetValue();
return result * _factor
```

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

DemoPGO

JIT – Altri punti...

- Tiering and Dynamic PGO
- Vectorization
- Branching
- Bounds Checking
- Constant Folding
- Non-GC Heap
- Zeroing
- Value Types
- Casting
- Peephole Optimizations



Performance Improvements in .NET 8
Performance Improvements in ASP.NET Core 8

.NET 9 ?

PGO: Enable profiled casts by default #96597

.NET 8 v .NET 9 LINQ Performance improvements

```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.17763.5328/1809/October2018Update/Redstone5
AMD EPYC 7763, 1 CPU, 4 logical and 2 physical cores
.NET SDK 9.0.100-alpha.1.24062.11
  [Host] : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
  .NET 8 : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
  .NET 9 : .NET 9.0.0 (9.0.24.6126), X64 RyuJIT AVX2
```

Method	Runtime	Length	Mean	Error	StdDev	Ratio
Min	.NET 8.0	50	12.773 ns	0.0666 ns	0.0591 ns	baseline
Min	.NET 9.0	50	6.905 ns	0.0406 ns	0.0380 ns	-46%
Max	.NET 8.0	50	12.129 ns	0.0132 ns	0.0117 ns	baseline
Max	.NET 9.0	50	7.660 ns	0.0210 ns	0.0197 ns	-37%
Count	.NET 8.0	50	6.722 ns	0.0089 ns	0.0075 ns	baseline
Count	.NET 9.0	50	2.086 ns	0.0030 ns	0.0026 ns	-69%
ElementAt	.NET 8.0	50	14.940 ns	0.0270 ns	0.0253 ns	baseline
ElementAt	.NET 9.0	50	4.335 ns	0.0089 ns	0.0079 ns	-71%
SequenceEqual	.NET 8.0	50	25.836 ns	0.0205 ns	0.0182 ns	baseline
SequenceEqual	.NET 9.0	50	7.989 ns	0.0943 ns	0.0882 ns	-69%

Conclusioni

- .NET 8 è un passaggio essenziale
- Se siete già nel trend cominciato con .NET 5/6, .NET 8 dovrebbe essere un in-place upgrade (sperabilmente) poco doloroso con innumerevoli vantaggi

Thanks!