C# PERFORMANCE BENCHMARKS

Gain insights in your code performance

TABLE OF CONTENTS

01

GOALS

GOALS

- Learn how to benchmark small code snippets
- Gain empirical knowledge from examples
- Challenge your believes of what is performant

02

BENCHMARKDOTNET

BENCHMARKDOTNET

- Open source library for .NET (Link)
- Easy to setup and use
- Runs benchmarks in their own processes
- Finds parameters to create reliable & reproducable results
- Used by .NET runtime and 16k+ projects

BENCHMARKDOTNET

```
[Config(typeof(DefaultBenchmarkConfig))]
simple enough (5%)   Johannes Deml
public class _TestBenchmark
{
    [Benchmark]
    simple enough (5%)   Johannes Deml
    public int InstantReturn()
    {
        return 0;
    }
}
```

Optional: Define a config that should be used

Add Benchmark Attribute above methods you want to benchmark

BENCHMARKDOTNET

```
[Params(params values:10, 10_000)]
5 usages
public int ArraySize { get; set; }

private byte[] sourceArray;
private byte[] destinationArray;

[GlobalSetup]
simple enough (5%)  Johannes Deml
public void PrepareBenchmark()
{
    sourceArray = ValuesGenerator.Array<byte>(ArraySize);
    destinationArray = new byte[ArraySize];
}
```

Define parameters to test with different input

Setup will be run once before all benchmarks

BENCHMARKDOTNET

```
simple enough (5%)   1 usage   1 override   Johannes Deml
protected virtual Job DefineBaseJob()
{
    return Job.Default
        .WithUnrollFactor(16)
        .WithWarmupCount(1)
        .WithIterationTime(TimeInterval.FromMilliseconds(250))
        .WithMinIterationCount(15)
        .WithMaxIterationCount(20)
        .WithGcServer(true)
        .WithGcConcurrent(true)
        .WithGcForce(true)
        .WithPlatform(Platform.AnyCpu); // Job
}
```

You can configure how you want to run the test

Define good enough values for development and quick tests

BENCHMARKDOTNET

```
simple enough (5%)   1 usage   Johannes Deml
public static class Program
{
    simple enough (5%)   Johannes Deml
    private static int Main(string[] args)
    {
        ManualConfig config = ManualConfig.CreateMinimumViable();

        return BenchmarkSwitcher
            .FromAssembly(typeof(Program).Assembly) // BenchmarkSwitcher
            .Run(args, config) // IEnumerable<Summary>
            .ToExitCode(); // int
    }
}
```

Handy helper to get a selection screen before any benchmark runs

MicroBenchmarks — MicroBenchmarks ◂ dotnet run --configuration Release --framework net6.0 — 113×38

```
johannes.deml@MB-10001724 MicroBenchmarks % clear

johannes.deml@MB-10001724 MicroBenchmarks % dotnet run --configuration Release --framework net6.0
Available Benchmarks:
  #0  _TestBenchmark
  #1  CallerInformationAttributesBenchmark
  #2  CollectionAddEntriesBenchmark
  #3  CollectionContainsBenchmark
  #4  CollectionInstantiationBenchmark
  #5  ConcurrentCollectionsBenchmark
  #6  ConditionalLoggingBenchmark
  #7  CopyBytesBenchmark
  #8  HashGenerationBenchmark
  #9  HmacGenerationBenchmark
  #10 LambdaBenchmark
  #11 LoopArraySumComparisonBenchmark
  #12 LoopListSumComparisonBenchmark
  #13 PauseAccuracyBenchmark
  #14 RangeBenchmark
  #15 SortingBenchmark
  #16 StringComparisonBenchmark
  #17 StringConcatenationBenchmark
  #18 StringSearchBenchmark
  #19 ThreadingBenchmark
  #20 TimeMeasurementBenchmark
  #21 UdpBenchmark


You should select the target benchmark(s). Please, print a number of a benchmark (e.g. `0`) or a contained benchm
ark caption (e.g. `_TestBenchmark`).
If you want to select few, please separate them with space ` ` (e.g. `1 2 3`).
You can also provide the class name in console arguments by using --filter. (e.g. `--filter '*_TestBenchmark*'`).
```

Run with Release configuration

Nice list of the switcher to run one or multiple benchmarks

MicroBenchmarks.CollectionContainsBenchmark-report-github.md

```
BenchmarkDotNet=v0.13.5, OS=ubuntu 22.04
AMD EPYC 7702P, 1 CPU, 4 logical and 4 physical cores
.NET SDK=6.0.116
  [Host]     : .NET 6.0.16 (6.0.1623.17701), X64 RyuJIT AVX2
  Job-XSGSJK : .NET 6.0.16 (6.0.1623.17701), X64 RyuJIT AVX2

Platform=AnyCpu  Runtime=.NET 6.0  Concurrent=True
Force=True  Server=True  Version=1.1.0
OS=Linux 5.15.0-48-generic #54-Ubuntu SMP Fri Aug 26 13:26:29 UTC 2022  DateTime=06/23/2023
06:41:33  SystemTag=Ubuntu VPS
```

Method	CollectionLength	Mean	Error	StdDev
ArrayForEachLoop	**10**	**3.172 ns**	**0.0856 ns**	**0.0801 ns**
ArrayLinqContains	10	6.547 ns	0.1338 ns	0.1045 ns
ListForEachLoop	10	8.130 ns	0.1971 ns	0.2631 ns
ListFindIndex	10	32.111 ns	0.4702 ns	0.3927 ns
ListExists	10	33.648 ns	0.6840 ns	0.7877 ns
ListContains	10	5.259 ns	0.1254 ns	0.1173 ns
EnumerableForEachLoop	10	53.297 ns	1.0635 ns	1.0922 ns
ReadOnlyListContains	10	8.720 ns	0.1214 ns	0.1076 ns

Information on the Hardware/Software used

Results with Mean, Error & Standard Deviation

You can also export to csv, xml, html, png or own custom formats

03

PERFORMANCE COMPARISONS

CAVEATS

- .NET != Unity
- Unity builds -> IL2CPP conversion
- Tests only ran on Ubuntu 22.04 with .NET 8
- Memory allocations left out

SETUP

- Virtual Private Server (kvm virtualization)
- 4 dedicated CPU cores (AMD EPYC 7702P)
- Ubuntu 22.04.3 LTS
- .NET 8 (8.0.23.53103)
- Running some other stuff (e.g. gitlab, Postgresql)
- Running with default BenchmarkDotNet settings
- Full Specs

CONDITIONAL LOGGING

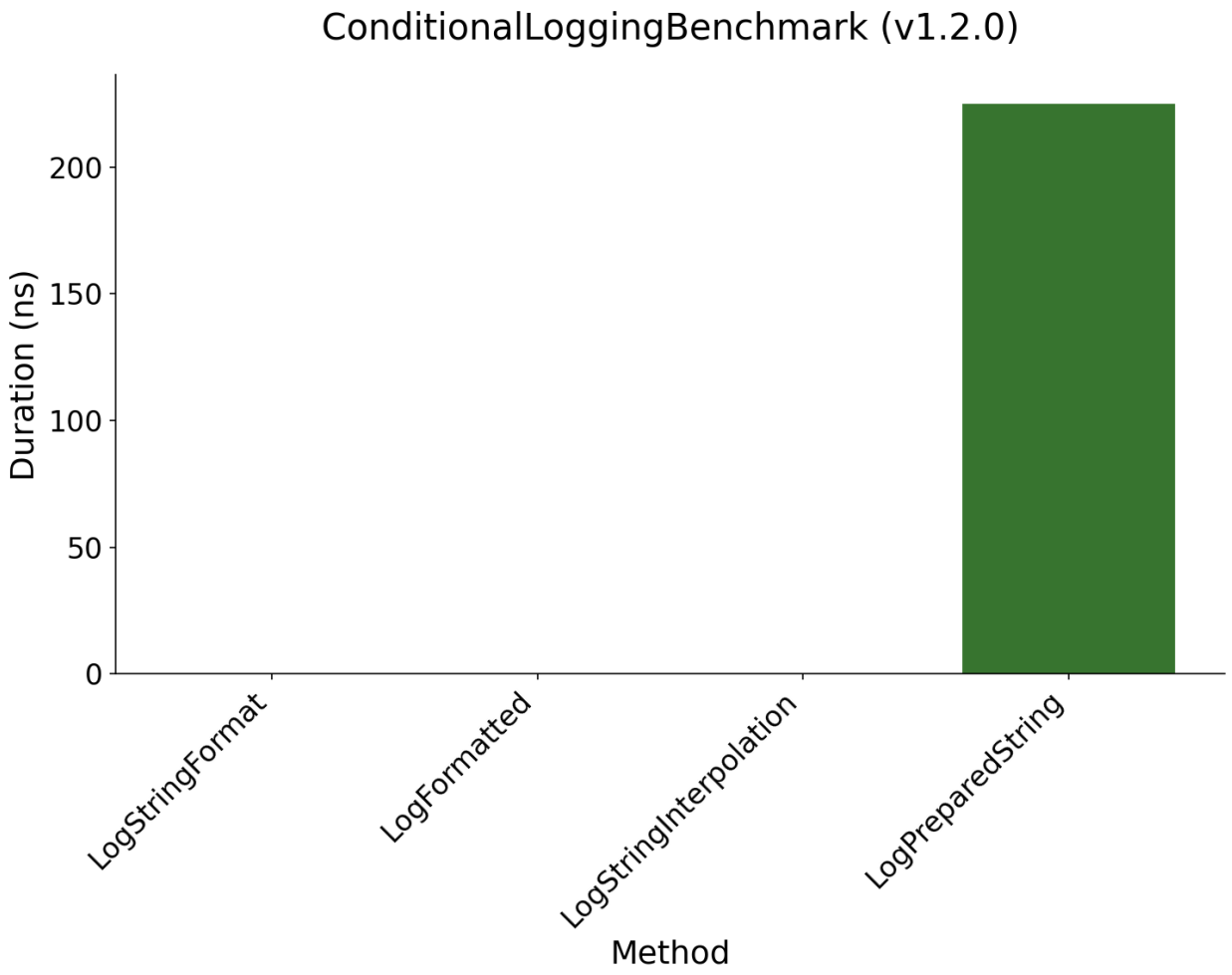
```
[Conditional( conditionString: "ALWAYS_FALSE_CONDITION")]
simple enough (5%)   3 usages   Johannes Deml
private void Log(string message)
{
    Console.Write(message);
}

[Conditional( conditionString: "ALWAYS_FALSE_CONDITION")]
simple enough (5%)   1 usage   Johannes Deml
private void LogFormat(string message, object param0, object param1, object param2)
{
    Console.Write(message, param0, param1, param2);
}
```

Will be stripped out of the code on compile time

CONDITIONAL LOGGING

- LogFormatted: `LogFormat(string, a, b, c);`
- LogStringInterpolation: `Log($”{a}{b}{c}”);`
- LogStringFormat: `Log(string.Format(string, a, b, c));`
- LogPreparedString: `string prepared=$”{a}{b}{c}”; Log(prepared);`

CONDITIONAL LOGGING



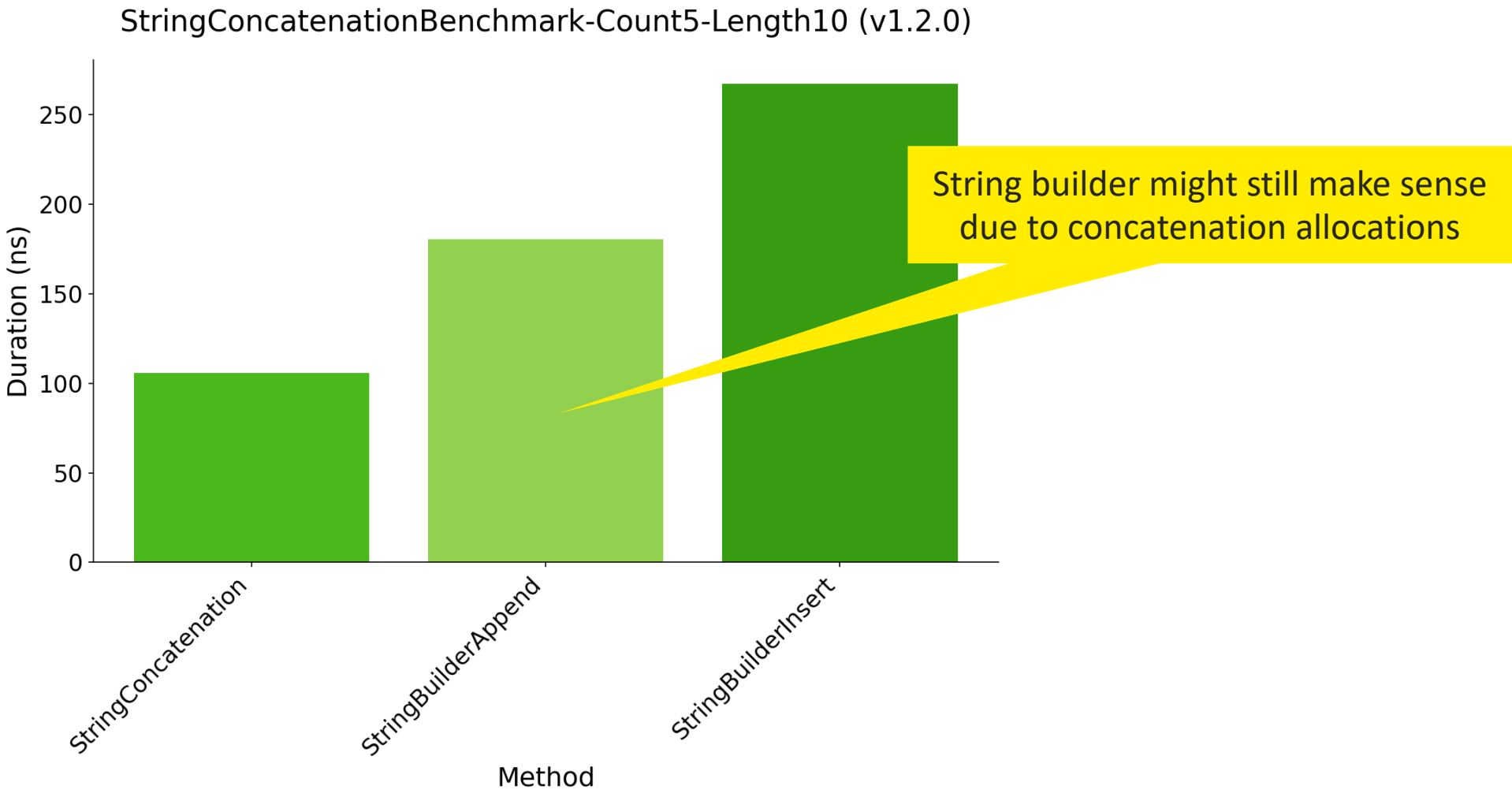
STRINGS - CONCATENATION

```
[Params(params values:5, 100)]
1 usage
public int StringCount { get; set; }

[Params(params values:10, 1_000)]
1 usage
public int StringLength { get; set; }
```

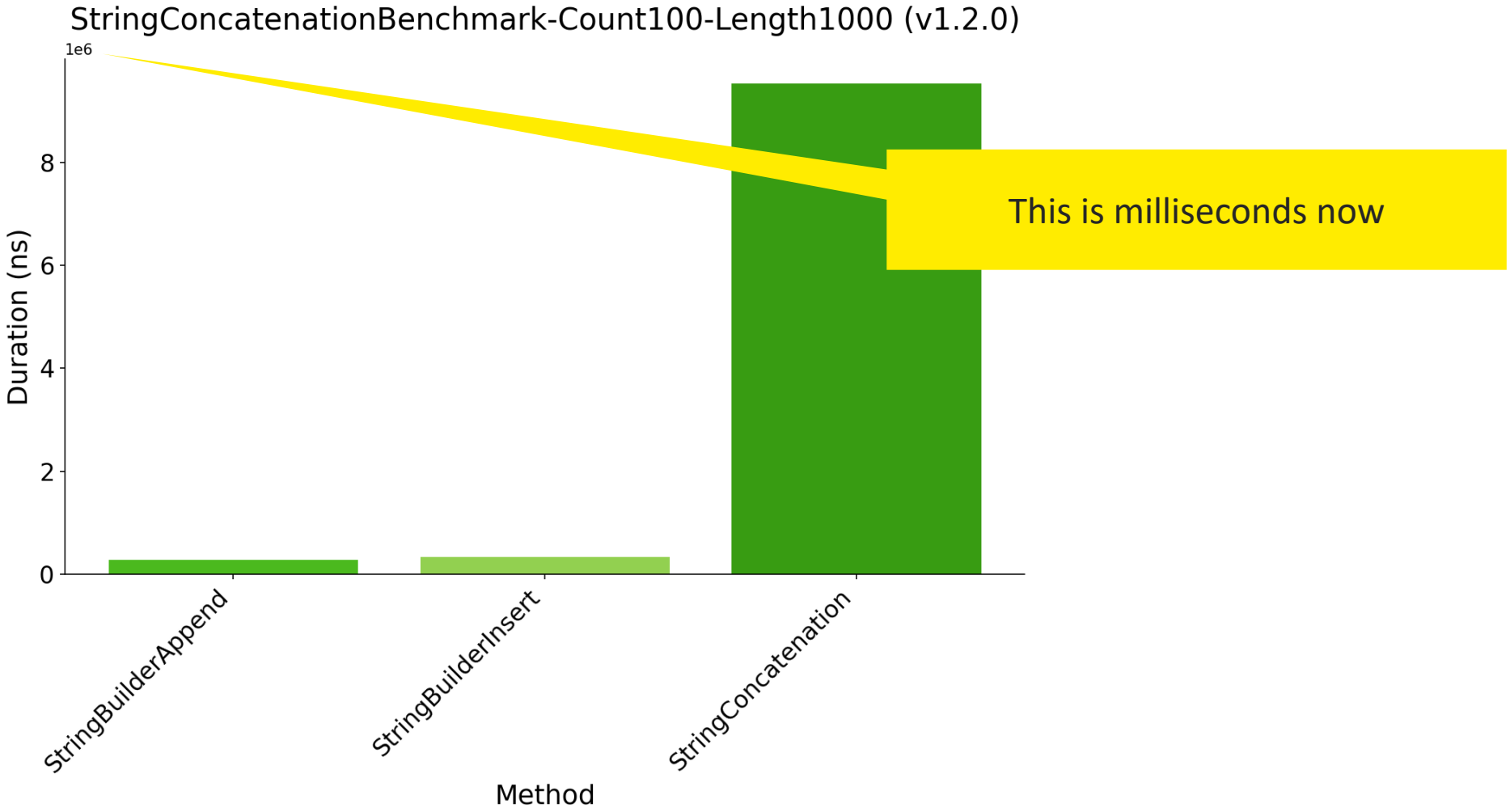
STRINGS- CONCATENATION

- StringConcatenation: `a += b;`
- StringBuilderAppend: `sb.Append(b);`
- StringBuilderInser: `sb.Insert(b);`

STRINGS - CONCATENATION



STRINGS - CONCATENATION



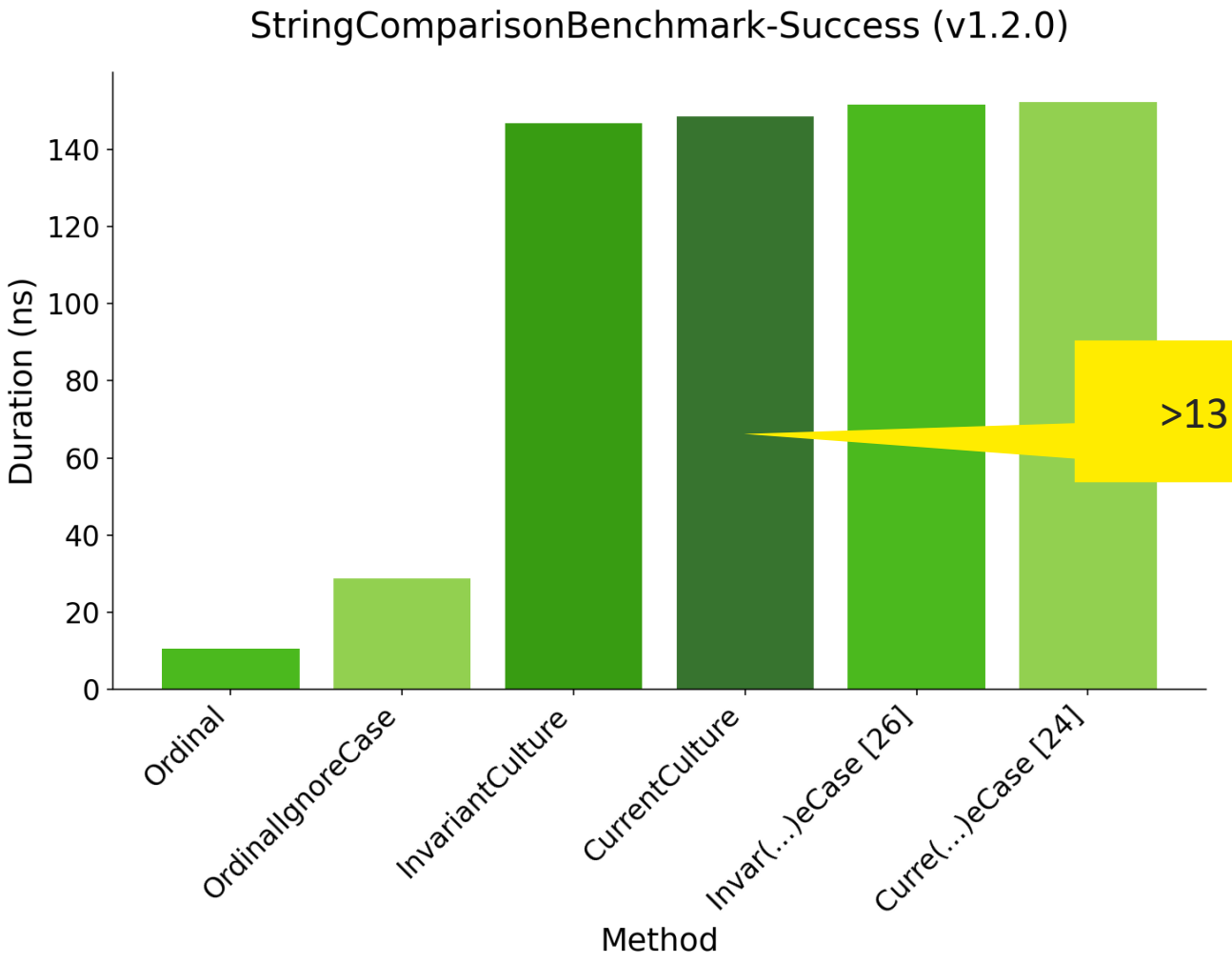
STRINGS - STRINGCOMPARISON

```
[Params(params values:StringComparison.Ordinal, StringComparison.OrdinalIgnoreCase,
    StringComparison.InvariantCulture, StringComparison.InvariantCultureIgnoreCase,
    StringComparison.CurrentCulture, StringComparison.CurrentCultureIgnoreCase)]
2 usages
public StringComparison Comparison { get; set; }

[Params(params values:100)]
2 usages
public int SearchStringLength { get; set; }
private const int TotalStringLength = 500;
```

STRINGS - STRINGCOMPARISON

- StartsWithStringSuccess: `stringA.StartsWith(a, Comparison);`

STRINGS - STRINGCOMPARISON



>13 times slower than Ordinal

STRINGS - SEARCH

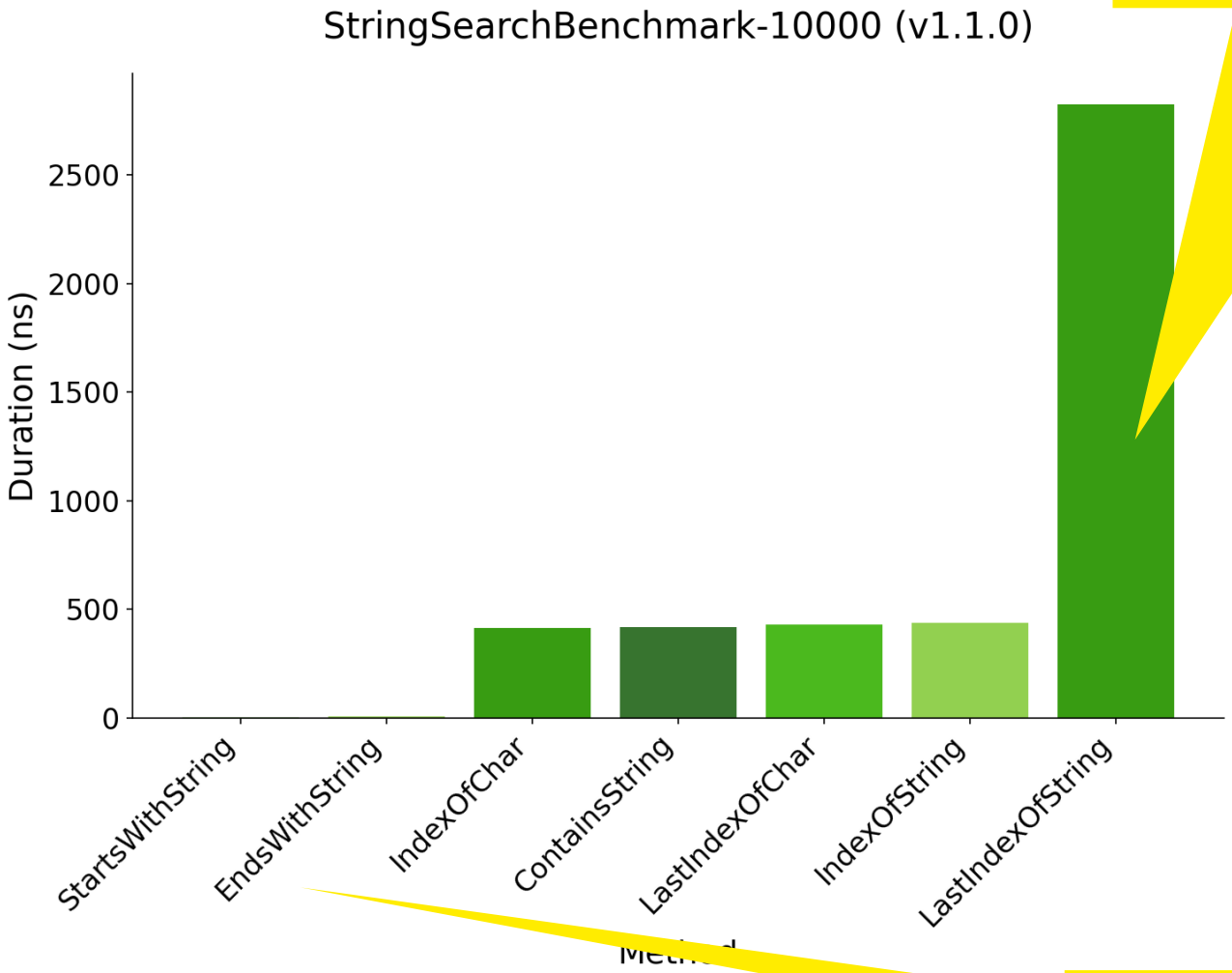
```
[Params(params values:10, 10_000)]
2 usages
public int LengthToTarget { get; set; }

private const char TargetChar = '|';
private const string TargetString = "|";
private string stringData;
```

STRINGS - SEARCH

- ContainsString: `stringData.Contains(s);`
- IndexOfString: `stringData.IndexOf(s, Ordinal);`
- LastIndexOfString: `stringData.LastIndexOf(s, Ordinal);`
- IndexOfChar: `stringData.IndexOf(c, Ordinal);`
- LastIndexOfChar: `stringData.LastIndexOf(c, Ordinal);`

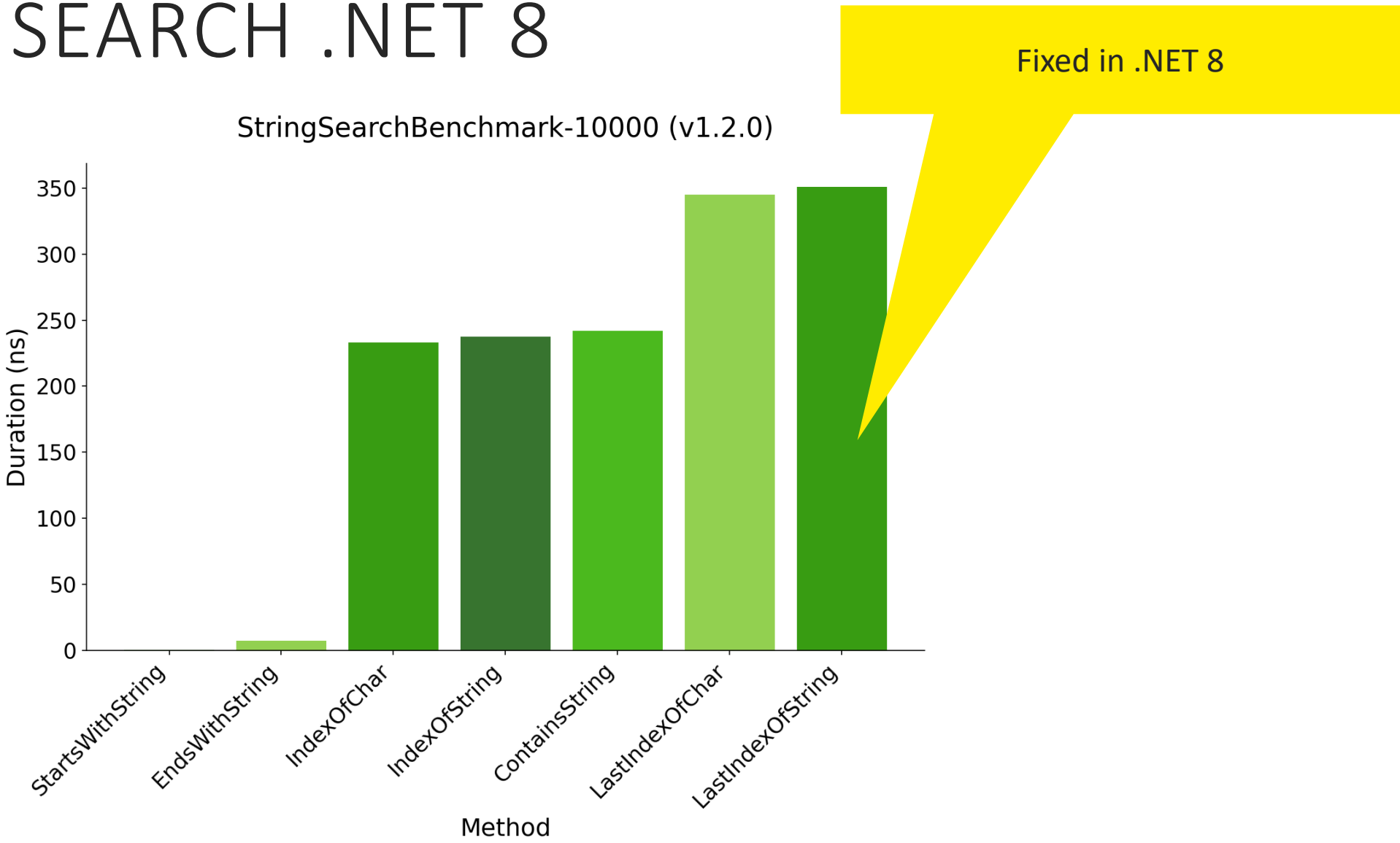
STRINGS – SEARCH .NET 6

>5 times than LastIndexOfChar



Good to see, that they don’t scale with string length

STRINGS – SEARCH .NET 8



ARRAY SUM LOOP

```
// Needs to be a multiple of 4 to support ForLoopUnroll4
[Params(params values: 100, 100_000)]
1 usage
public int ArraySize { get; set; }
```

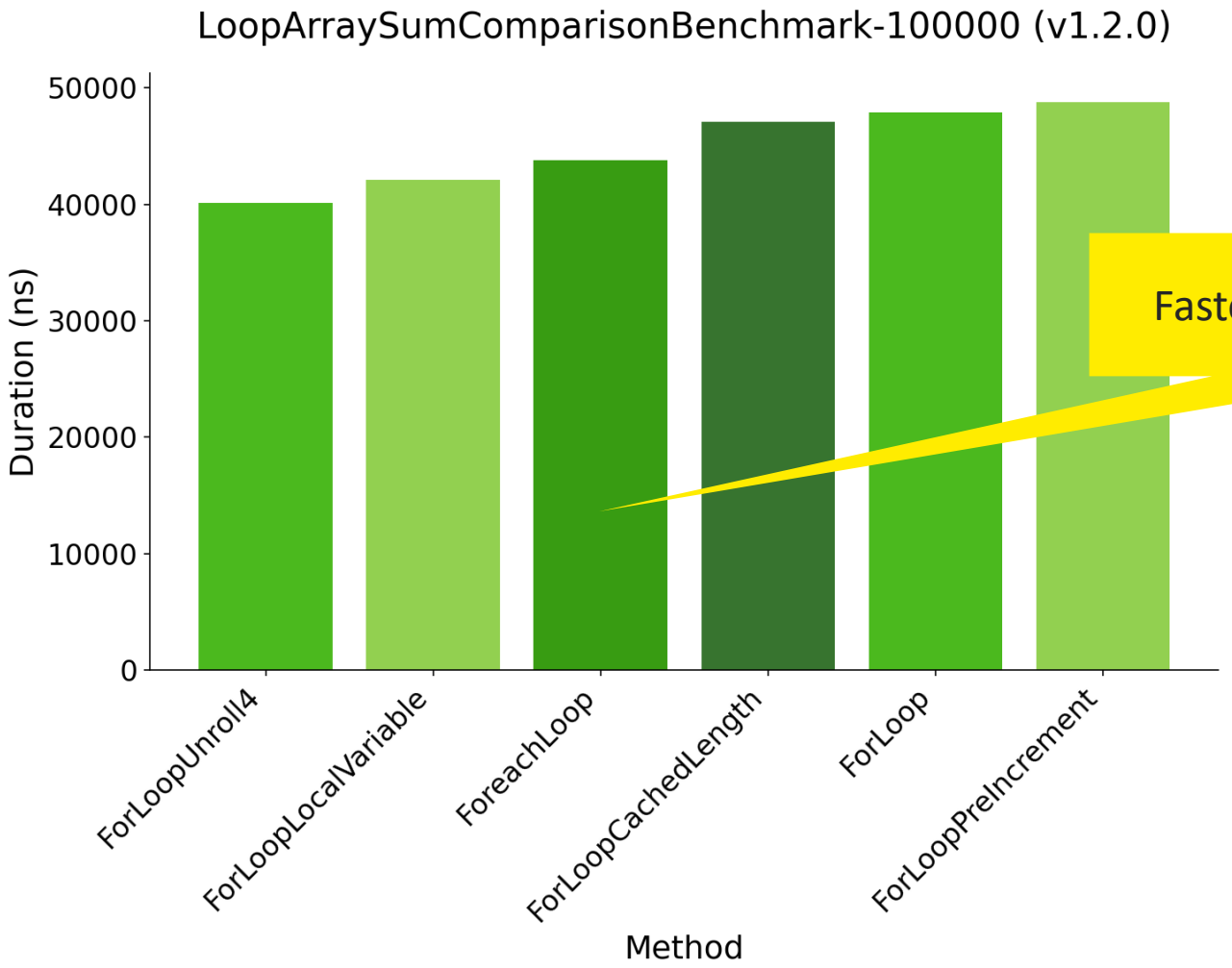
ARRAY SUM LOOP

- ForLoop: Normal for loop
- ForLoopCachedLength: for loop check with local length var
- ForLoopLocalVariable: assign local var with array, then loop over that
- ForLoopUnroll4: Always sum up 4 values and the jump by 4 in loop
- ForeachLoop: Normal foreach loop
- LinqSum: `array.Sum(b => b);`
- LinqAggregate: `array.Aggregate(0, (sum, b) => sum + b);`

ARRAY SUM LOOP

~5 times slower than ForEachLoop

ARRAY SUM LOOP



Faster than “normal” for loop 🤔

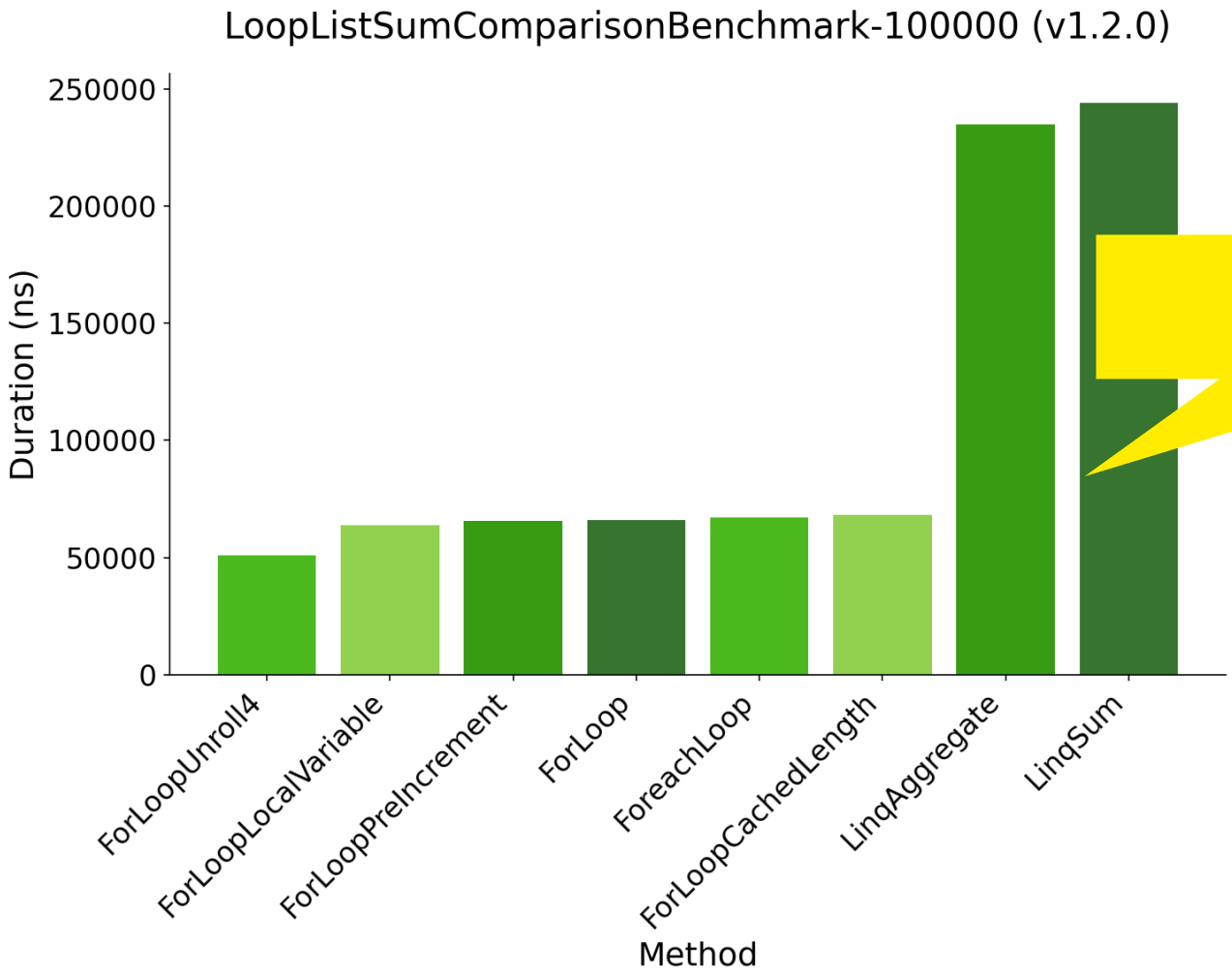
LIST SUM LOOP

```
// Needs to be a multiple of 4 to support ForLoopUnroll4
[Params(params values:100, 100_000)]
1 usage
public int ListSize { get; set; }
```

LIST SUM LOOP

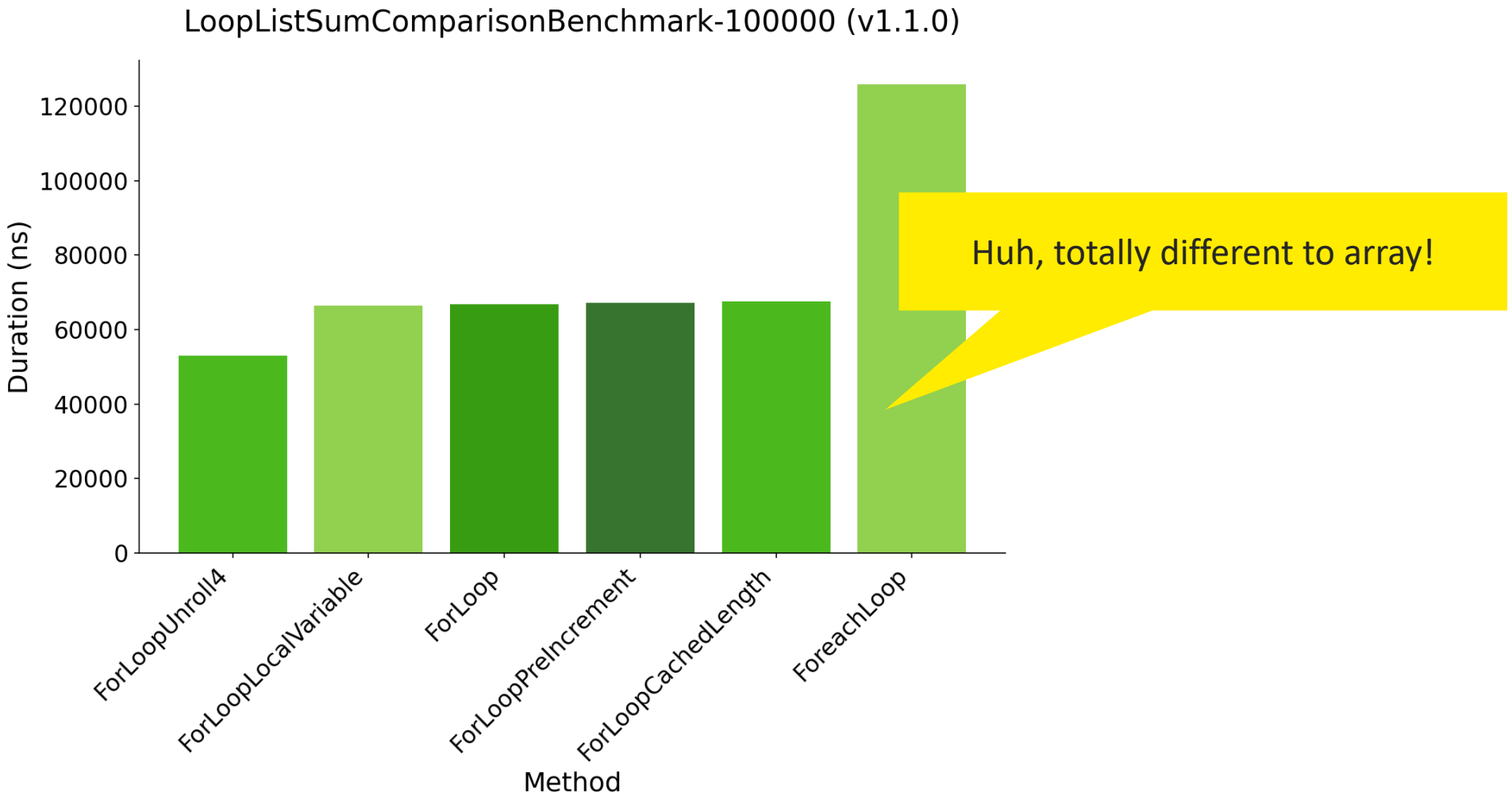
- ForLoop: Normal for loop
- ForLoopCachedLength: for loop check with local length var
- ForLoopLocalVariable: assign local var with list, then loop over that
- ForLoopUnroll4: Always sum up 4 values and the jump by 4 in loop
- ForeachLoop: Normal foreach loop
- LinqSum: `list.Sum(b => b);`
- LinqAggregate: `list.Aggregate(0, (sum, b) => sum + b);`

LIST SUM LOOP

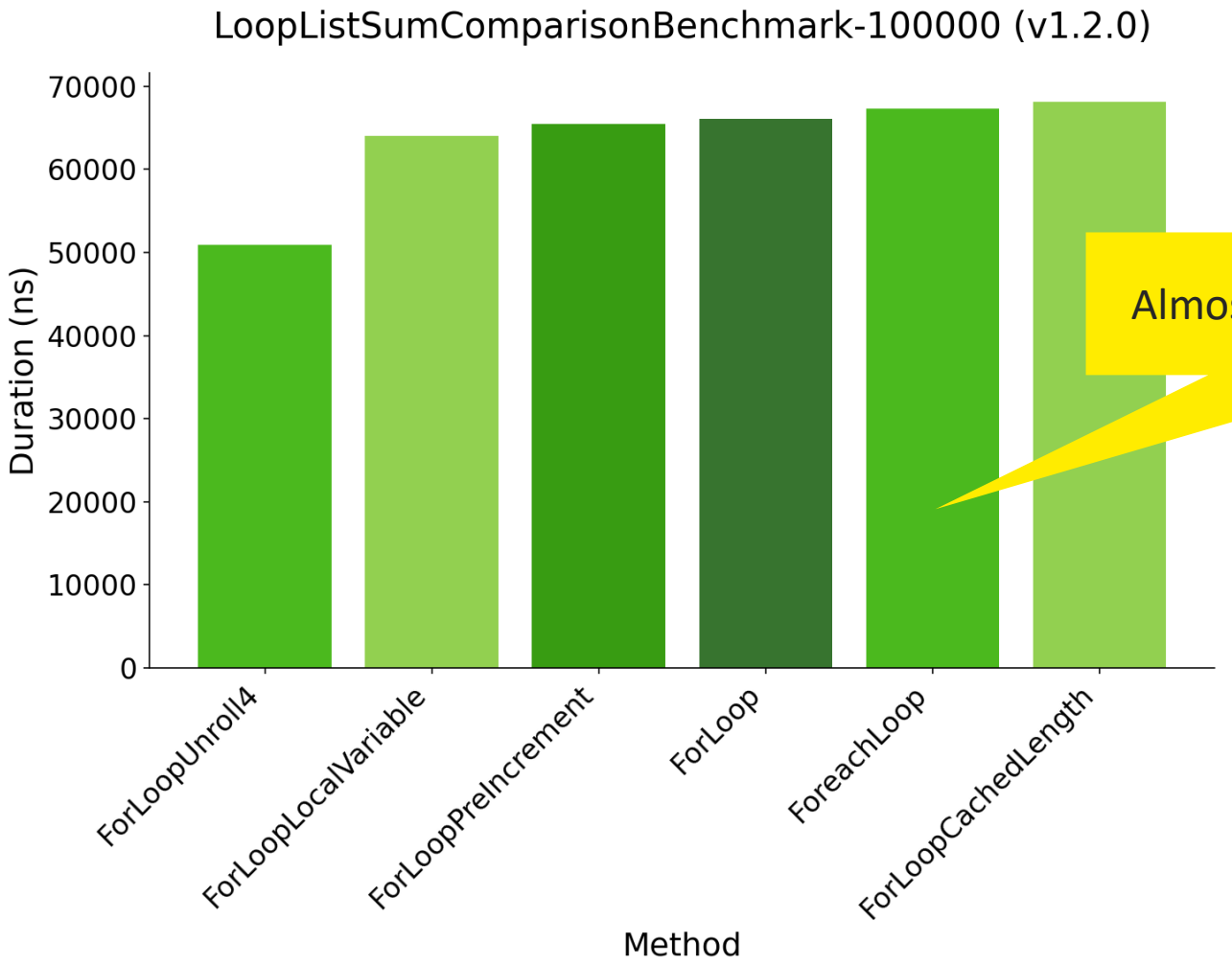


Again, no surprise here…

LIST SUM LOOP .NET 6



LIST SUM LOOP .NET 8



Almost as fast as for loop in .NET 8

ARRAY SUM LOOP

```
[Benchmark]
● simple enough (15%)   Johannes Deml
public int ForeachLoop()
{
    var sum = 0;
    foreach (var i:byte in data)
    {
        sum += i;
    }

    return sum;
}
```

IL Viewer

```
[Benchmark(106, "/Users/johannes.
public int ForeachLoop()
{
  int sum = 0;
  byte[] data = this.data;
  for (int index = 0; index < dat
  {
    byte i = data[index];
    sum += (int) i;
  }
  return sum;
}
```

LIST SUM LOOP

IL Viewer

```
[Benchmark]
simple enough (15%)   Johannes Deml
public int ForeachLoop()
{
    var sum = 0;
    foreach (var i:byte in data)
    {
        sum += i;
    }

    return sum;
}

[Benchmark]
simple enough (5%)   Johannes Deml
public int LinqSum()
{
    return data.Sum(b:byte => (int
}
```

```
[Benchmark(107, "/Users/johannes.deml/Docume
public int ForeachLoop()
{
  int sum = 0;
  List<byte>.Enumerator enumerator = this.da
  try
  {
    while (enumerator.MoveNext())
    {
      byte i = enumerator.Current;
      sum += (int) i;
    }
  }
  finally
  {
    enumerator.Dispose();
  }
  return sum;
}
```

COLLECTION CONTAINS

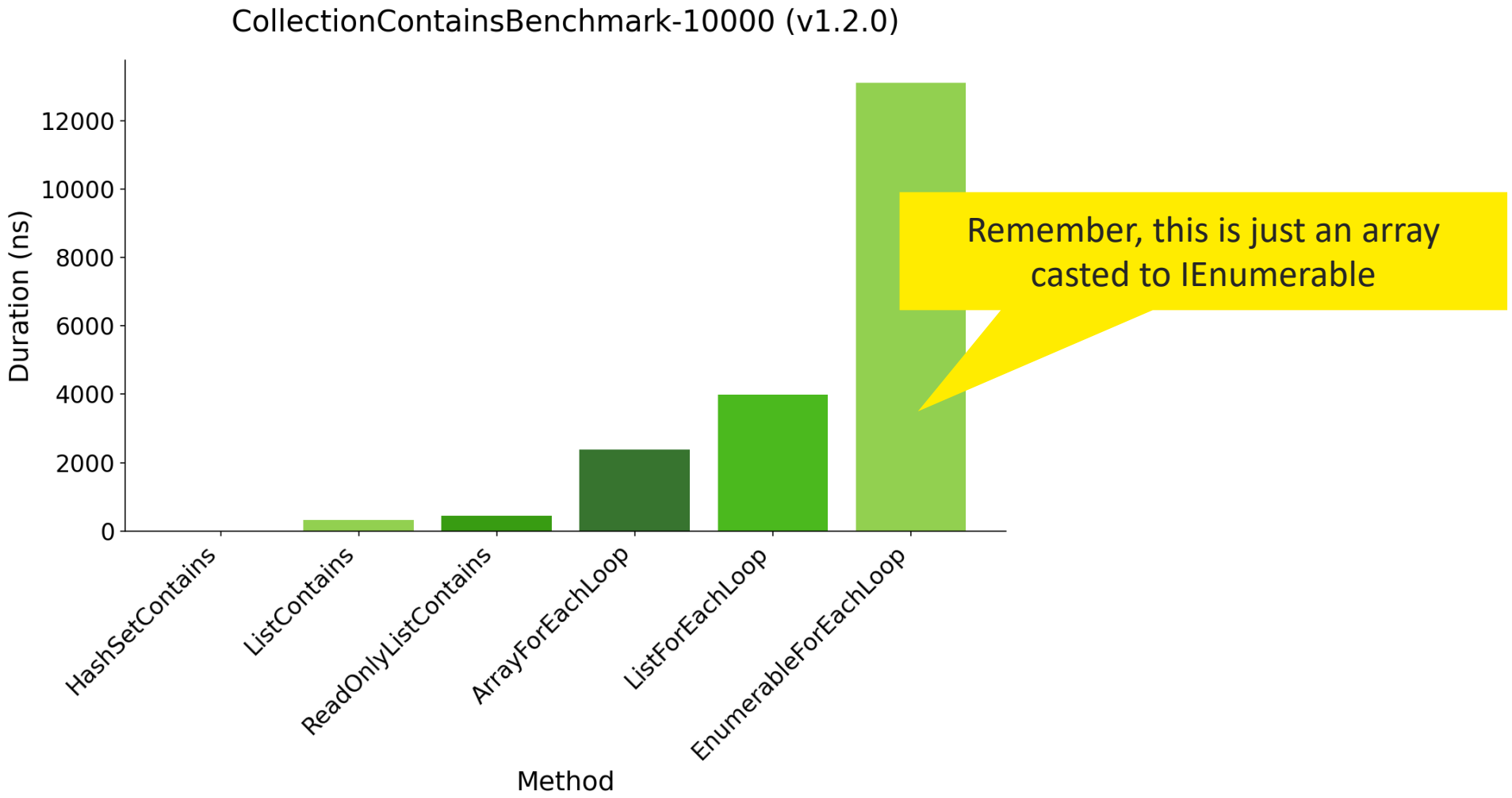
```
2 usages
[Params(params values:10, 10_000)] public int CollectionLength { get; set; }
```

```
array = ValuesGenerator.ArrayOfUniqueValues<int>(CollectionLength);
enumerable = array;
list = new List<int>(array);
readOnlyList = list;
```

COLLECTION CONTAINS

- ArrayForEach: `foreach(v in array) { if(v==i) {return true;} }`
- ListForEach: `foreach(v in list) { if(v==i) {return true;} }`
- EnumerableForEach: `foreach(v in enumerable) { if(v==i) {return true;} }`
- ListContains: `list.Contains(i);`
- ReadOnlyListContains: `readOnlyList.Contains(i);`
- HashSetContains: `hashSet.Contains(i);`

COLLECTION CONTAINS



04

TINKER YOURSELF

- BenchmarkDotNet
- MicrobenchmarksDotNet
- NetworkBenchmarkDotNet
- SerializationBenchmarkDotNet

All benchmarks of the presentation and more can be found here

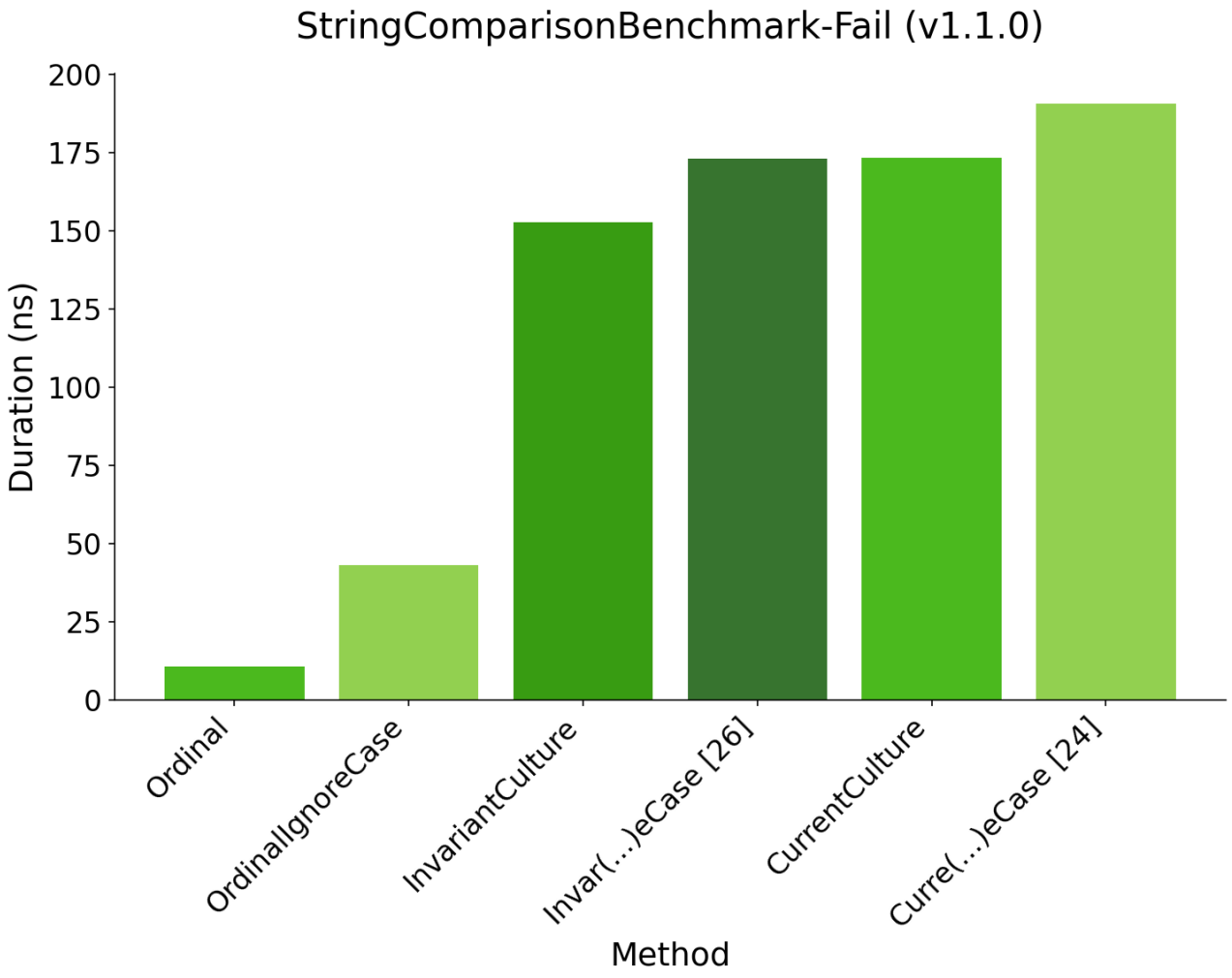
END OF PRESENTATION

Thank You

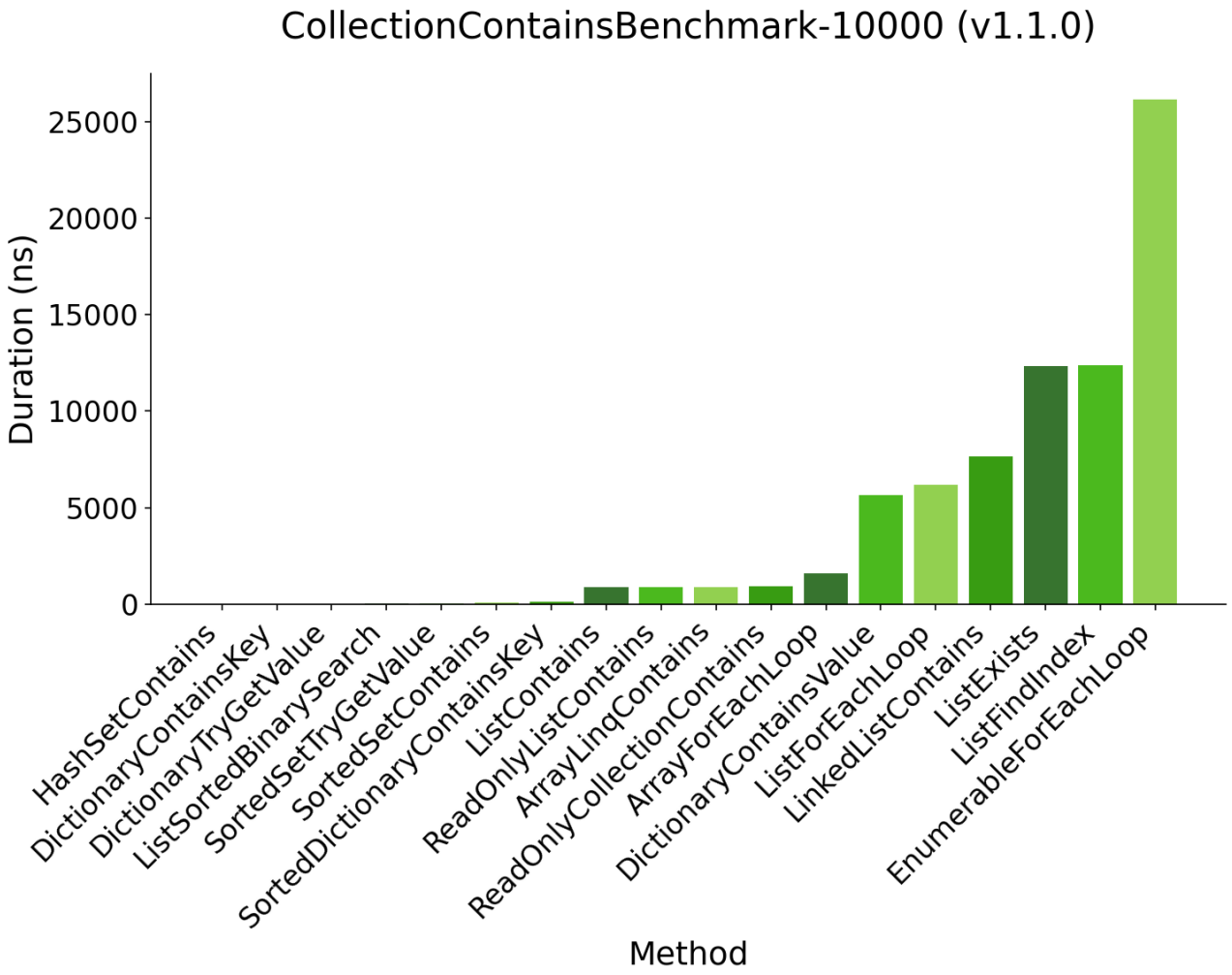
GET IN TOUCH

Johannes Deml
Frontend Developer
Sunrise Village

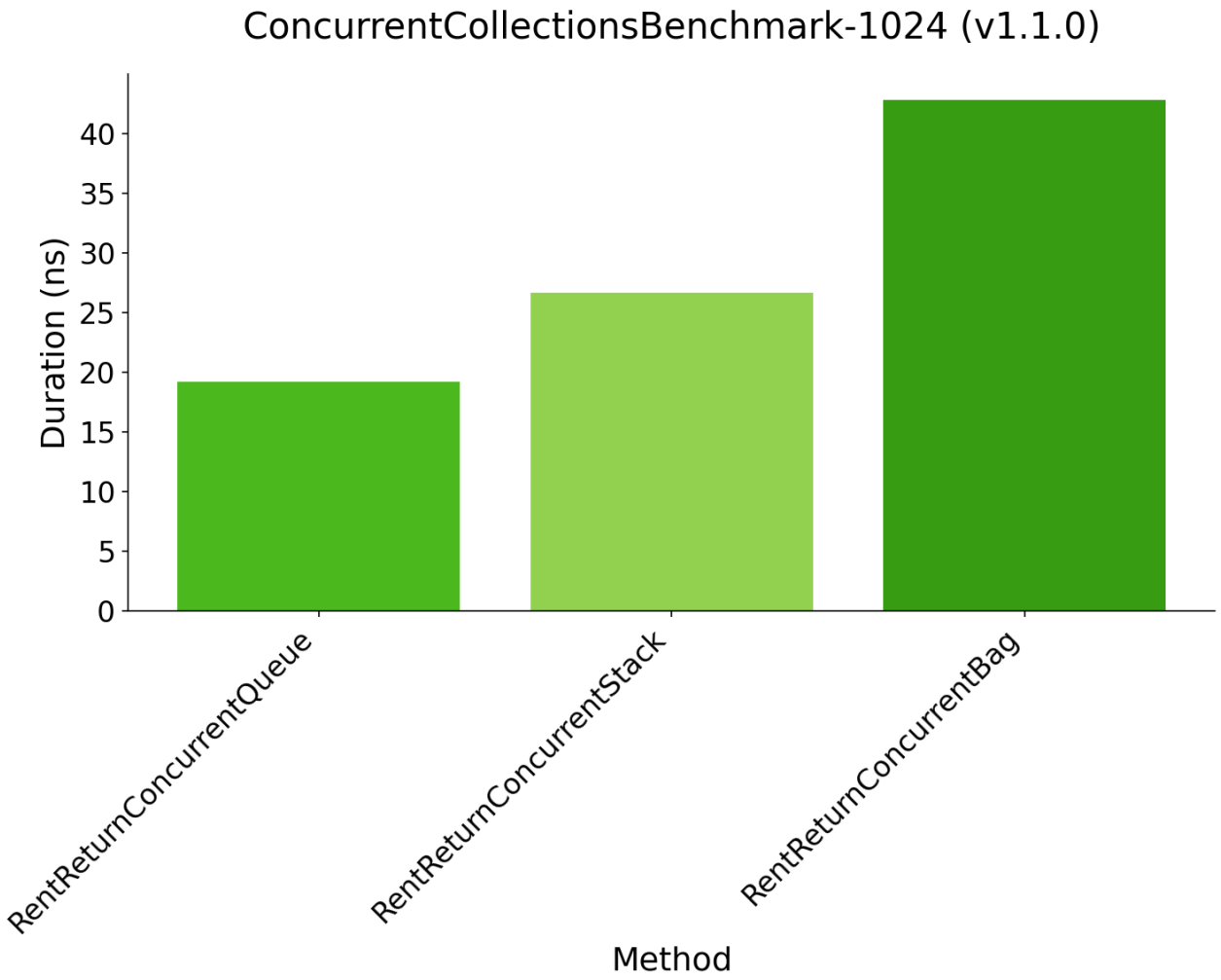
APPENDIX


StringComparisonBenchmark-Fail (v1.1.0)
Duration (ns)
200
175
150
125
100
75
50
25
0
Ordinal
OrdinalIgnoreCase
InvariantCulture
Invar(...)eCase [26]
CurrentCulture
Curre(...)eCase [24]
Method

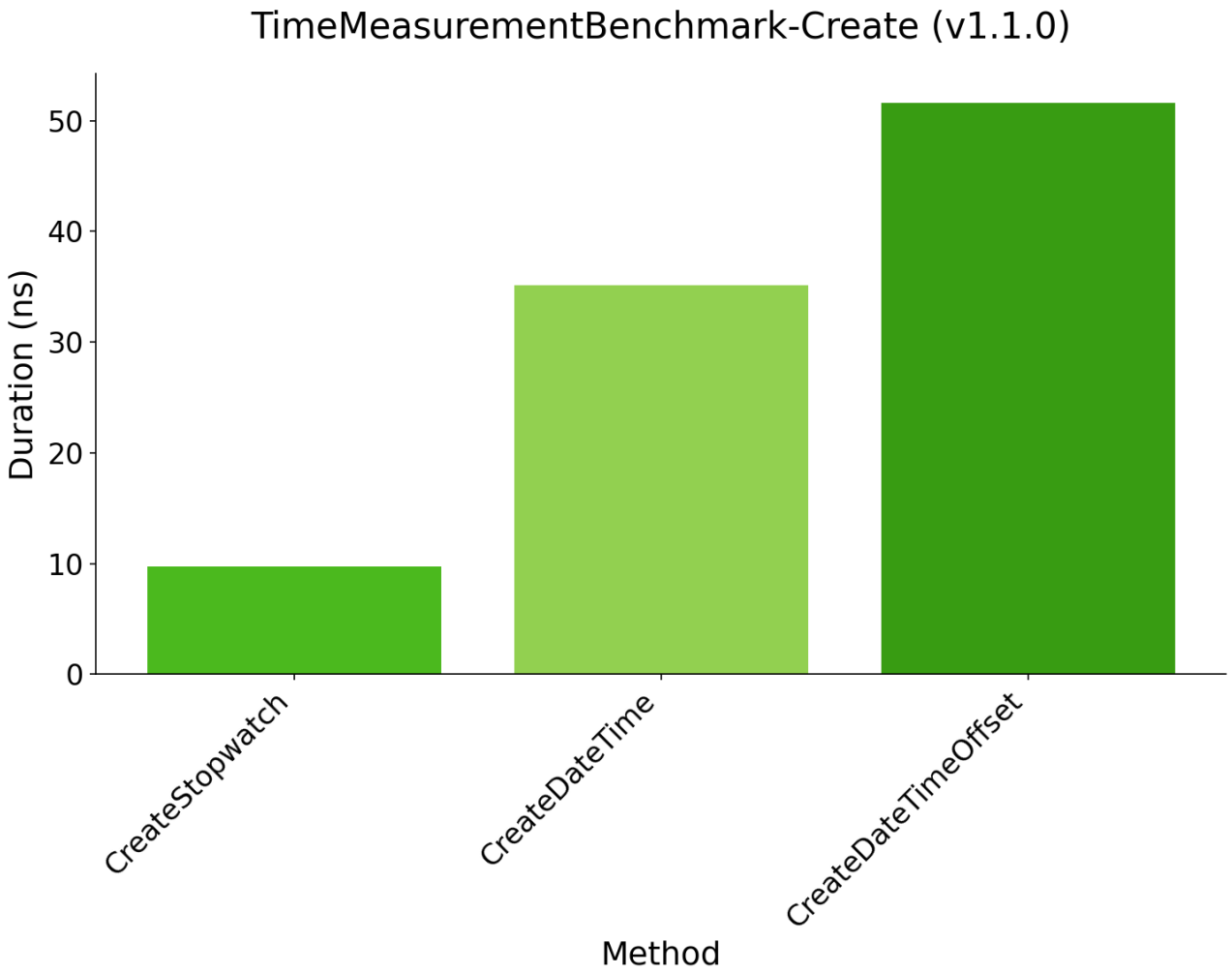
APPENDIX


CollectionContainsBenchmark-10000 (v1.1.0)
Duration (ns)
25000
20000
15000
10000
5000
0
HashSetContains
DictionaryContainsKey
DictionaryTryGetValue
ListSortedBinarySearch
SortedSetTryGetValue
SortedSetContains
SortedDictionaryContainsKey
ListContains
ReadOnlyListContains
ArrayLinqContains
ReadOnlyCollectionContains
ArrayForEachLoop
DictionaryContainsValue
ListForEachLoop
LinkedListContains
ListExists
ListFindIndex
EnumerableForEachLoop
Method

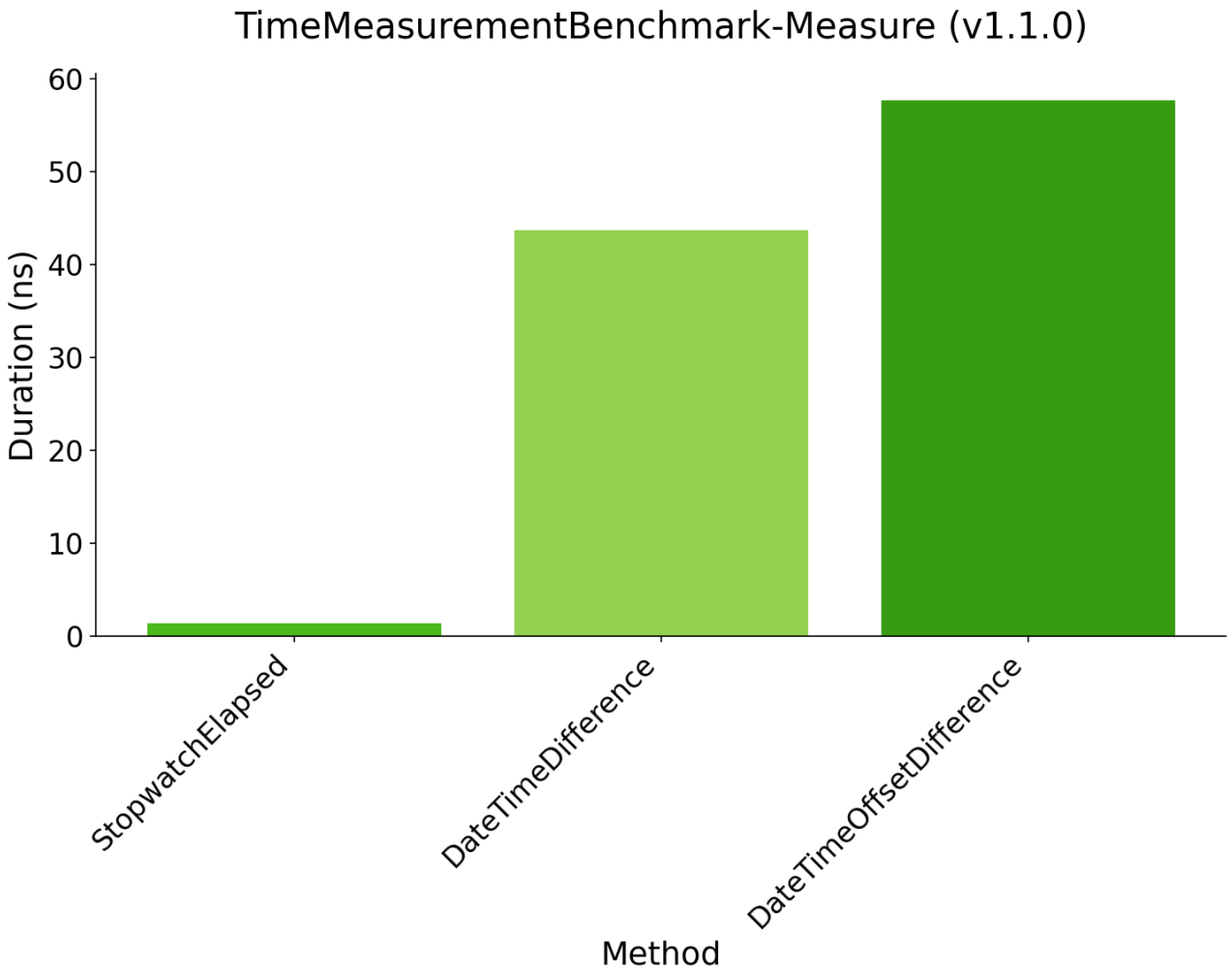
APPENDIX


ConcurrentCollectionsBenchmark-1024 (v1.1.0)
Duration (ns)
0
5
10
15
20
25
30
35
40
RentReturnConcurrentQueue
RentReturnConcurrentStack
RentReturnConcurrentBag
Method

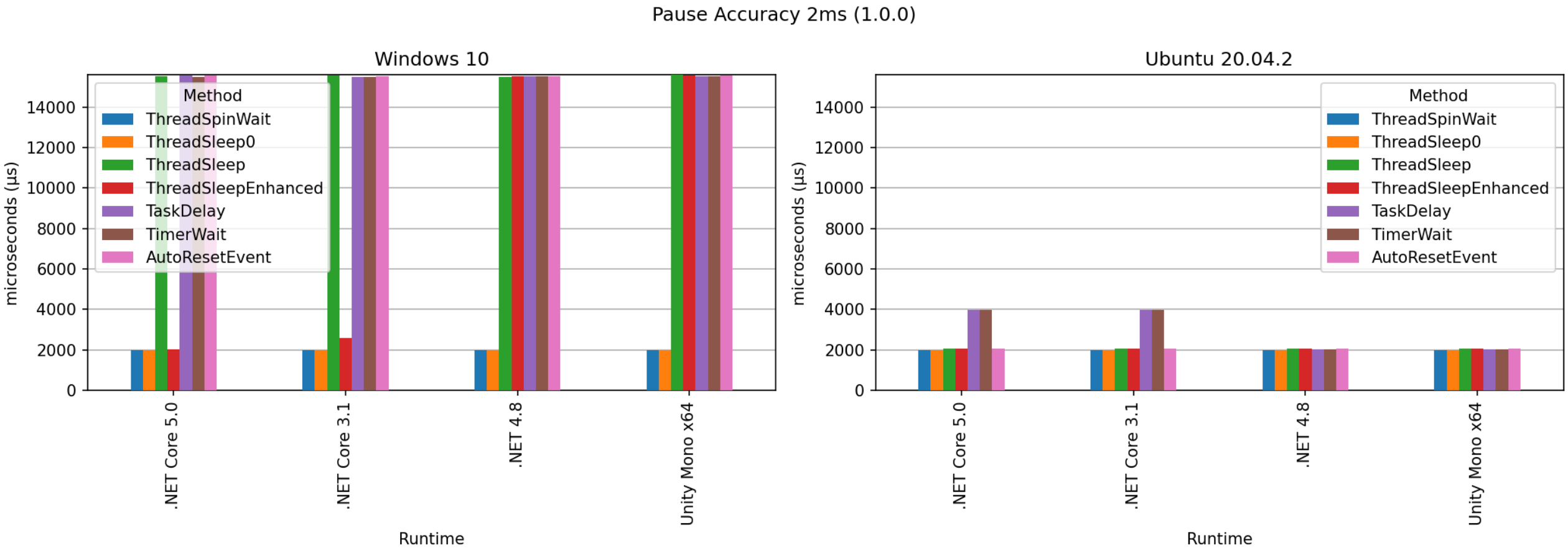
APPENDIX


TimeMeasurementBenchmark-Create (v1.1.0)
Duration (ns)
0
10
20
30
40
50
CreateStopwatch
CreateDateTime
CreateDateTimeOffset
Method

APPENDIX


TimeMeasurementBenchmark-Measure (v1.1.0)
Duration (ns)
60
50
40
30
20
10
0
StopwatchElapsed
DateTimeDifference
DateTimeOffsetDifference
Method

APPENDIX


Pause Accuracy 2ms (1.0.0)
Windows 10
Ubuntu 20.04.2
microseconds (µs)
0
2000
4000
6000
8000
10000
12000
14000
Method
ThreadSpinWait
ThreadSleep0
ThreadSleep
ThreadSleepEnhanced
TaskDelay
TimerWait
AutoResetEvent
.NET Core 5.0
.NET Core 3.1
.NET 4.8
Unity Mono x64
Runtime