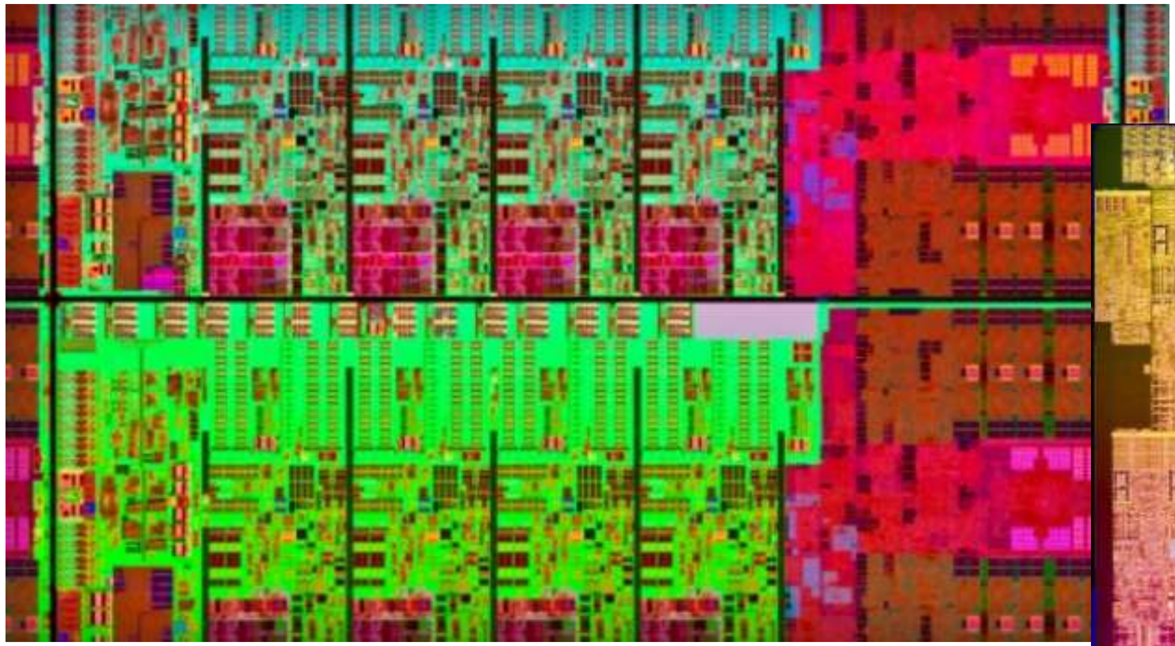
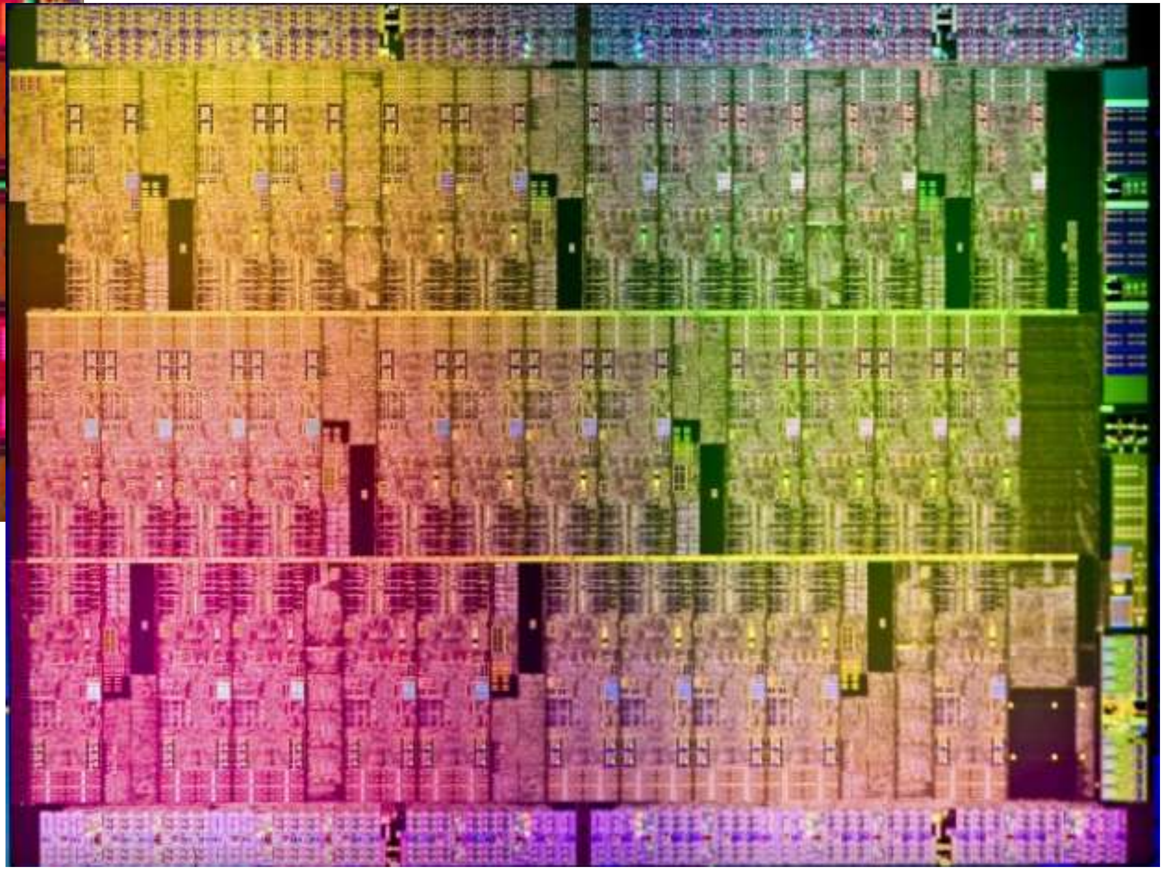
SIMD + NativeAOT + Dynamic PGO

Jiří Činčura (engineer, x̄ size) | Karel Zikmund (manager, tall)

A Modern Processor

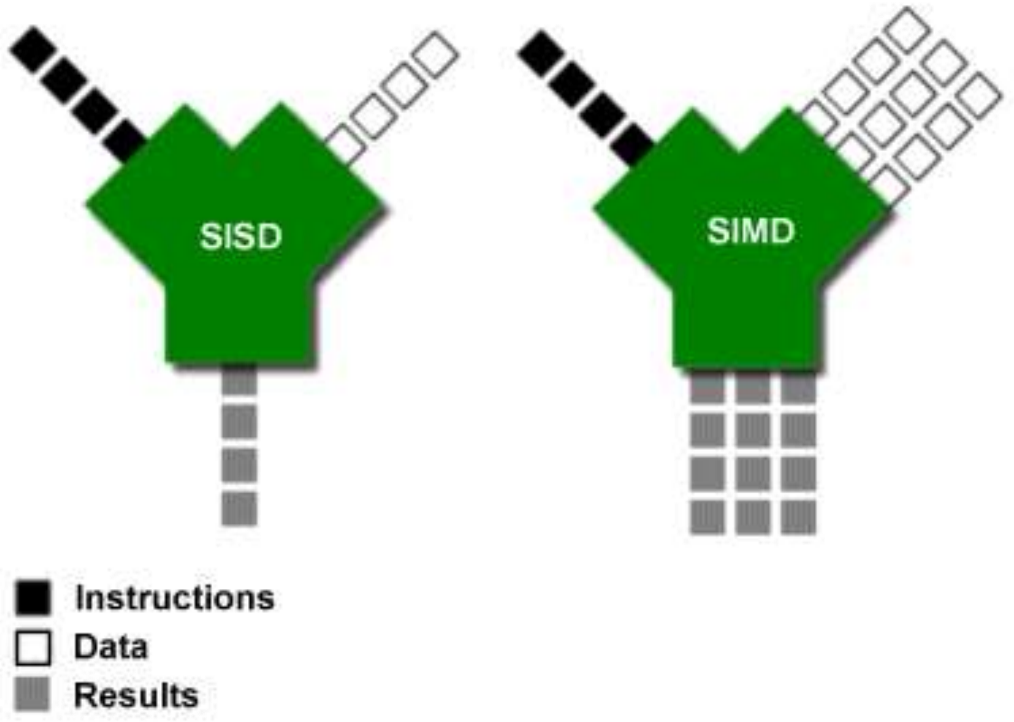
SISD

- Single Instruction Single Data
- Instruction Level Parallelism (ILP)
- Parallel processing

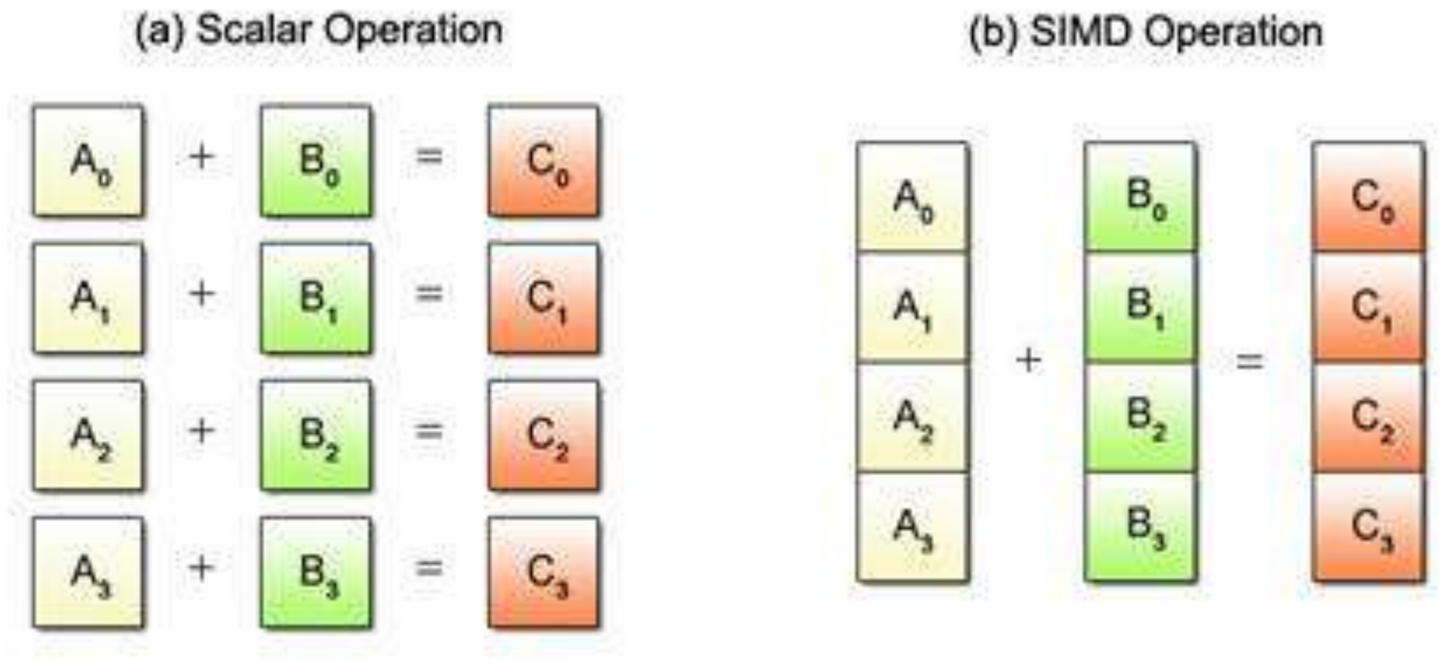
SIMD

- Single Instruction Multiple Data

SIMD



SIMD


(a) Scalar Operation
A_0 + B_0 = C_0
A_1 + B_1 = C_1
A_2 + B_2 = C_2
A_3 + B_3 = C_3
(b) SIMD Operation
A_0 A_1 A_2 A_3 + B_0 B_1 B_2 B_3 = C_0 C_1 C_2 C_3

NativeAOT

- Ahead-of-Time compilation
- JIT, startup time, memory/working set
- Benefits
    - Faster startup time
    - Smaller memory footprint
    - Self-contained executable
    - Restricted environments (i.e. iOS)
- Limitations
    - No dynamic loading
    - No runtime code gen (System.Reflection.Emit, limited reflection)
    - Bigger application binary

C# -> IL -> CPU

- C# code is compiled into IL (MSIL, CIL)
  - Stack based
  - Object oriented
- C# -> IL = Roslyn compiler (C# compiler)
- IL -> CPU = RyuJIT (JIT)
- Straightforward assembly code is not the fastest one
- JIT must generate great code and do it fast
  - 🤷‍♂️

PGO

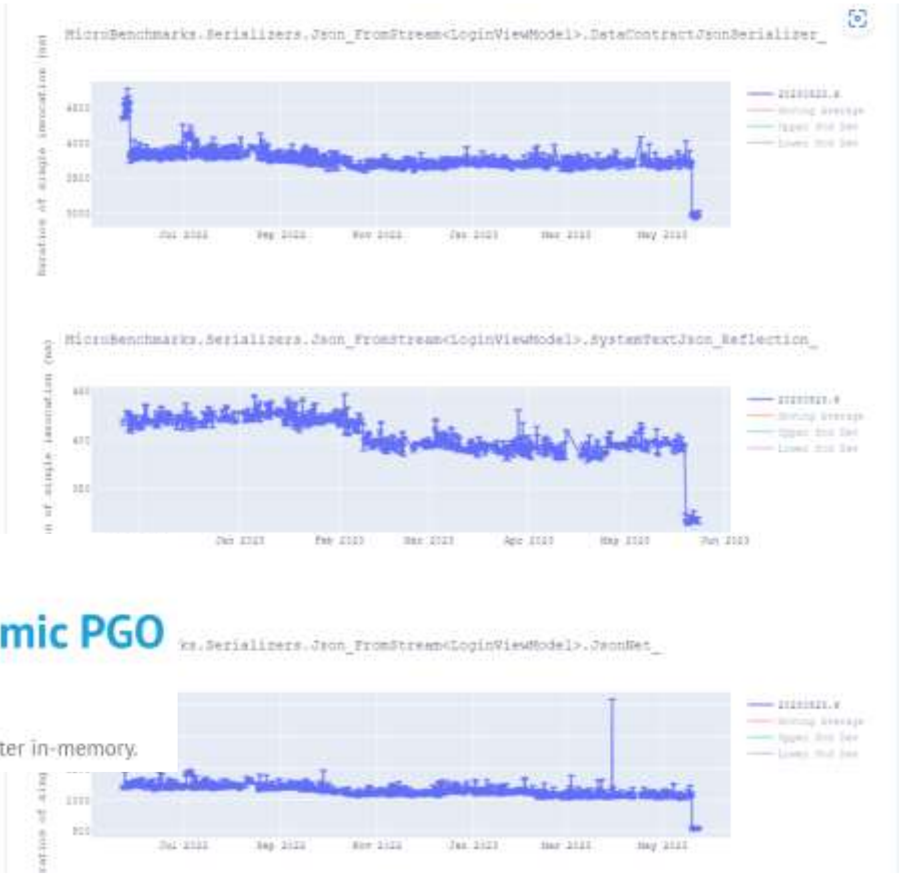
- Profile Guided Optimization
- Static
  - Collect data from representative run and store along the executable
  - Is the data up-to-date?
- Dynamic
  - In-proc, no training or special builds
  - Uses Tiered Compilation by instrumenting code in initial tiers
  - Collected data used later for better or more optimization

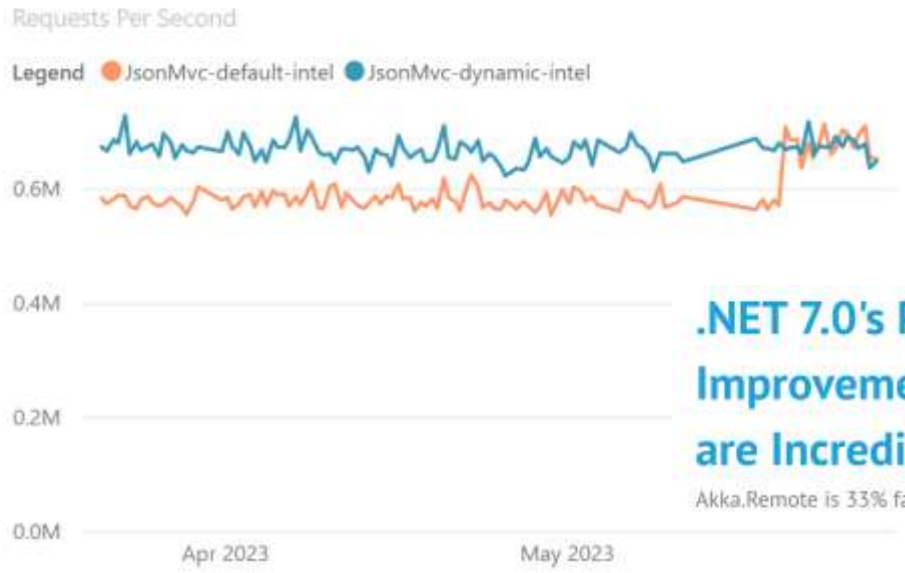
Benefits

- About 15% or up

[Perf] Linux/x64: 1590 Improvements on 5/19/2023 3:32:16 PM #18171

Open performanceautofiler opened this issue 2 days ago · 73 comments





.NET 7.0's Performance Improvements with Dynamic PGO are Incredible

Akka.Remote is 33% faster, Akka.NET v1.5 is 75% faster in-memory.

Key Optimizations

- Guarded Devirtualization (GDV)
  - At virtual and interface call sites, introduce tests for specific types
  - If the test succeeds, we know exactly which method will be called
    - Also try and inline the method
    - If the method is on a value class, inline the unboxing stub and the method
      - If source is a box attempt to optimize away the box too
  - If the test fails, just do the normal virtual / interface call
- .NET 8: extends GDV to handle some delegate invokes as well
- Opt-in: Multiple guesses GDV

GDV

```
void RegisterUser(IUserService service, User user)
{
    service.Register(user); // virtual call
}
```

```
void RegisterUser(IUserService service, User user)
{
    CORINFO_HELP_CLASSPROFILE32(service.GetType());
    service.Register(user);
}
```

```
void RegisterUser(IUserService service, User user)
{
    if (service is UserServiceImpl impl)
        impl.Register(user); // direct call, can be inlined
    else
        service.Register();  // still virtual (fallback)
}
```

GDV

```
void RegisterUser(IUserService service, User user)
{
    service.Register(user); // virtual call
}
```

```
void RegisterUser(IUserService service, User user)
{
    CORINFO_HELP_CLASSPROFILE32(user.GetType());
    service.Register(user);
}
```

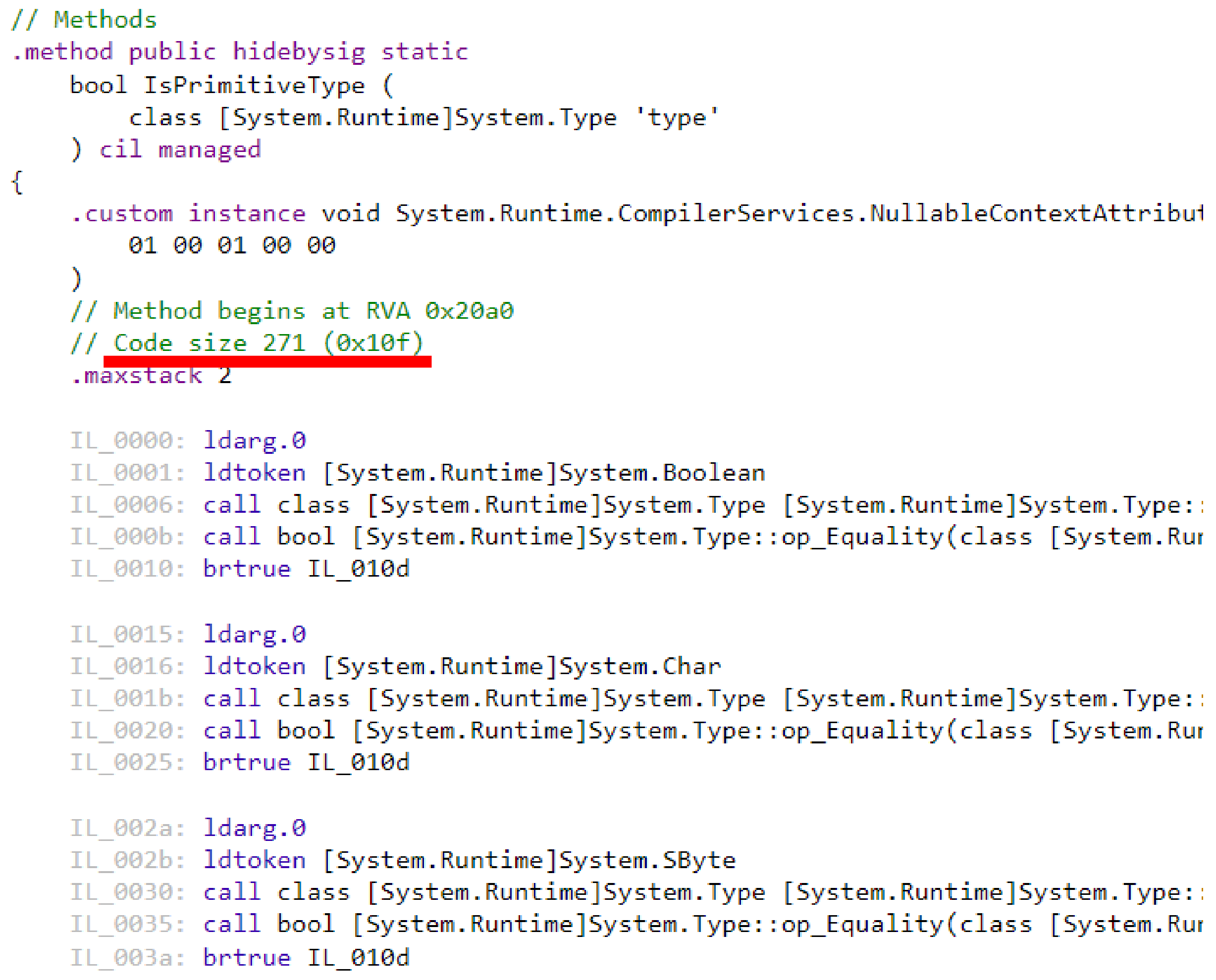
```
void RegisterUser(IUserService service, User user)
{
    if (service is UserServiceImpl impl)
        impl.Register(user);
    else if (service is GenericUserService1<User> impl)
        impl.Register(user);
    else if (service is GenericUserService2<User> impl)
        impl.Register(user);
    else
        service.Register();
}
```

Profile-Driven Inlining

- Use profile data to ensure key methods are inlined
- Relaxed thresholds for IL size and number of basic blocks
- Waste less energy on (semi-) cold call sites

Profile-Driven Inlining

```
bool IsPrimitiveType(Type type) =>
    type == typeof(bool)   ||
    type == typeof(char)   ||
    type == typeof(sbyte)  ||
    type == typeof(byte)   ||
    type == typeof(short)  ||
    type == typeof(ushort) ||
    type == typeof(int)    ||
    type == typeof(uint)   ||
    type == typeof(long)   ||
    type == typeof(ulong)  ||
    type == typeof(float)  ||
    type == typeof(double) ||
    type == typeof(nint)   ||
    type == typeof(nuint);
```



```
// Methods
.method public hidebysig static
    bool IsPrimitiveType (
        class [System.Runtime]System.Type 'type'
    ) cil managed
{
    .custom instance void System.Runtime.CompilerServices.NullableContextAttribut
        01 00 01 00 00
    )
    // Method begins at RVA 0x20a0
    // Code size 271 (0x10f)
    .maxstack 2

    IL_0000: ldarg.0
    IL_0001: ldtoken [System.Runtime]System.Boolean
    IL_0006: call class [System.Runtime]System.Type [System.Runtime]System.Type::
    IL_000b: call bool [System.Runtime]System.Type::op_Equality(class [System.Ru
    IL_0010: brtrue IL_010d

    IL_0015: ldarg.0
    IL_0016: ldtoken [System.Runtime]System.Char
    IL_001b: call class [System.Runtime]System.Type [System.Runtime]System.Type::
    IL_0020: call bool [System.Runtime]System.Type::op_Equality(class [System.Ru
    IL_0025: brtrue IL_010d

    IL_002a: ldarg.0
    IL_002b: ldtoken [System.Runtime]System.SByte
    IL_0030: call class [System.Runtime]System.Type [System.Runtime]System.Type::
    IL_0035: call bool [System.Runtime]System.Type::op_Equality(class [System.Ru
    IL_003a: brtrue IL_010d
```

Profile-Driven Inlining

```
for (int i = 0; i < 100; i++)
{
    Test<int, float>();
    Thread.Sleep(16);
}
```

```
[MethodImpl(MethodImplOptions.NoInlining)]
static bool Test<T1, T2>() =>
    IsPrimitiveType(typeof(T1)) &&
    IsPrimitiveType(typeof(T2));
```

```
static bool IsPrimitiveType(Type type) =>
    type == typeof(bool) ||
    type == typeof(char) ||
    type == typeof(sbyte) ||
    type == typeof(byte) ||
    type == typeof(short) ||
    type == typeof(ushort) ||
    type == typeof(int) ||
    type == typeof(uint) ||
    type == typeof(long) ||
    type == typeof(ulong) ||
    type == typeof(float) ||
    type == typeof(double) ||
    type == typeof(nint) ||
    type == typeof(nuint);
```

```
; Assembly listing for method Program:Test[int,float]():bool
; Tier-1 compilation
; No PGO data
       sub      rsp, 40
       mov      rcx, 0x11B802000B8 ; 'System.Int32'
       call     [Program:IsPrimitiveType(System.Type):bool]
       test     eax, eax
       je       SHORT G_M27198_IG05
       mov      rcx, 0x11B80205090 ; 'System.Single'
       call     [Program:IsPrimitiveType(System.Type):bool]
       nop
       add      rsp, 40
       ret
G_M27198_IG05:
       xor      eax, eax
       add      rsp, 40
       ret
; Total bytes of code 53
```

```
Inliner: too many IL bytes
```

Profile-Driven Inlining

```
for (int i = 0; i < 100; i++)
{
    Test<int, float>();
    Thread.Sleep(16);
}
```

```
[MethodImpl(MethodImplOptions.NoInlining)]
static bool Test<T1, T2>() =>
    IsPrimitiveType(typeof(T1)) &&
    IsPrimitiveType(typeof(T2));
```

```
static bool IsPrimitiveType(Type type) =>
    type == typeof(bool) ||
    type == typeof(char) ||
    type == typeof(sbyte) ||
    type == typeof(byte) ||
    type == typeof(short) ||
    type == typeof(ushort) ||
    type == typeof(int) ||
    type == typeof(uint) ||
    type == typeof(long) ||
    type == typeof(ulong) ||
    type == typeof(float) ||
    type == typeof(double) ||
    type == typeof(nint) ||
    type == typeof(nuint);
```

```
; Assembly listing for method Program:Test[int,float]():bool
; Tier-1 compilation
; Optimized with Dynamic PGO
       mov      eax, 1
       ret
; Total bytes of code 6
```

```
Inliner:
• Inline candidate has 13 foldable branches.
• Inline has 28 foldable intrinsics.
• Callsite has profile data: 1.0.
```

Instrumentation Overhead

- Dynamic PGO startup improvements in NET 8 · Issue #76969
- Sparse , scalable edge profiles enabled for all methods
- GDV random state now in TLS
- Scalable profile mode
- More cases where we bypass instrumentation
- Enable intrinsic expansion in Tier0

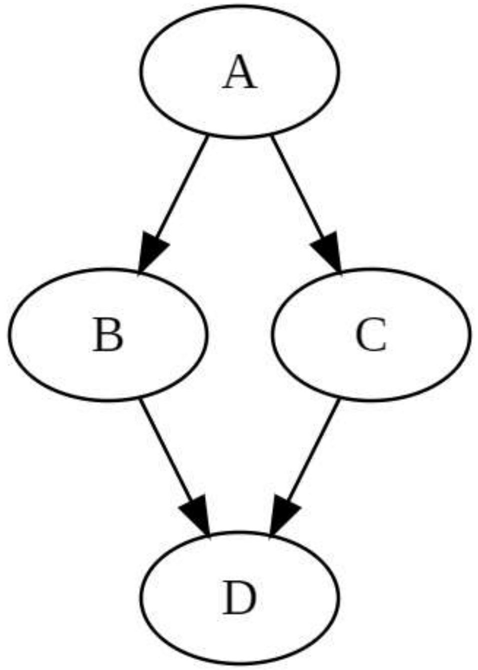
Class and Method Profiles

- *Reservoir sampling* used to create approximate histograms of target classes (for virtual/interface calls) and target methods (for indirect/delegate calls)
  - Fixed-sized table per site (currently 32 entries (was 8))
    - One global table per site
  - Each call adds entry to table, until table is full, then
    - Each call may randomly replace some table entry, with probability
    - This also keeps contention low
- When optimizing, this data is used to drive GDV, testing for the most likely outcome(s)

Profiling Blocks

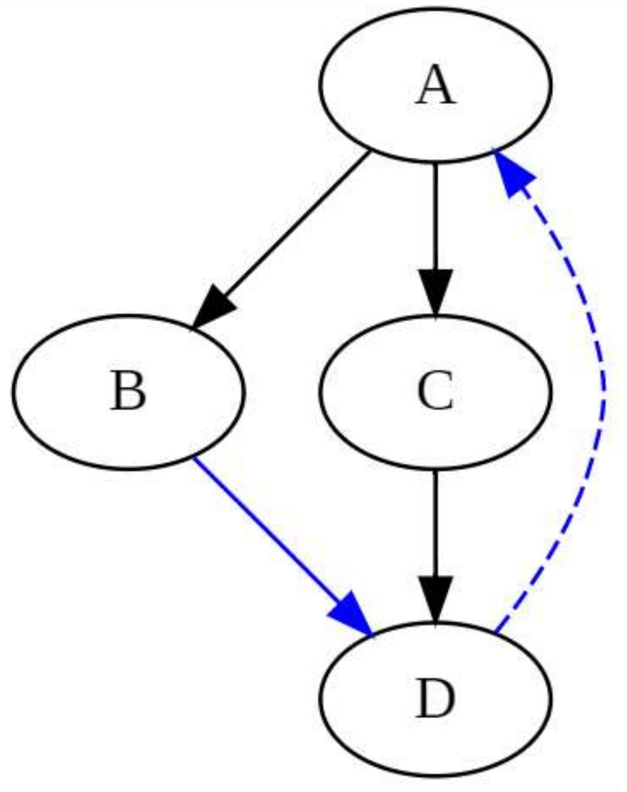
Dense

- Each block gets a counter
- Quite a bit of redundancy
    - Simple diamond: four blocks, two independent counts



Sparse

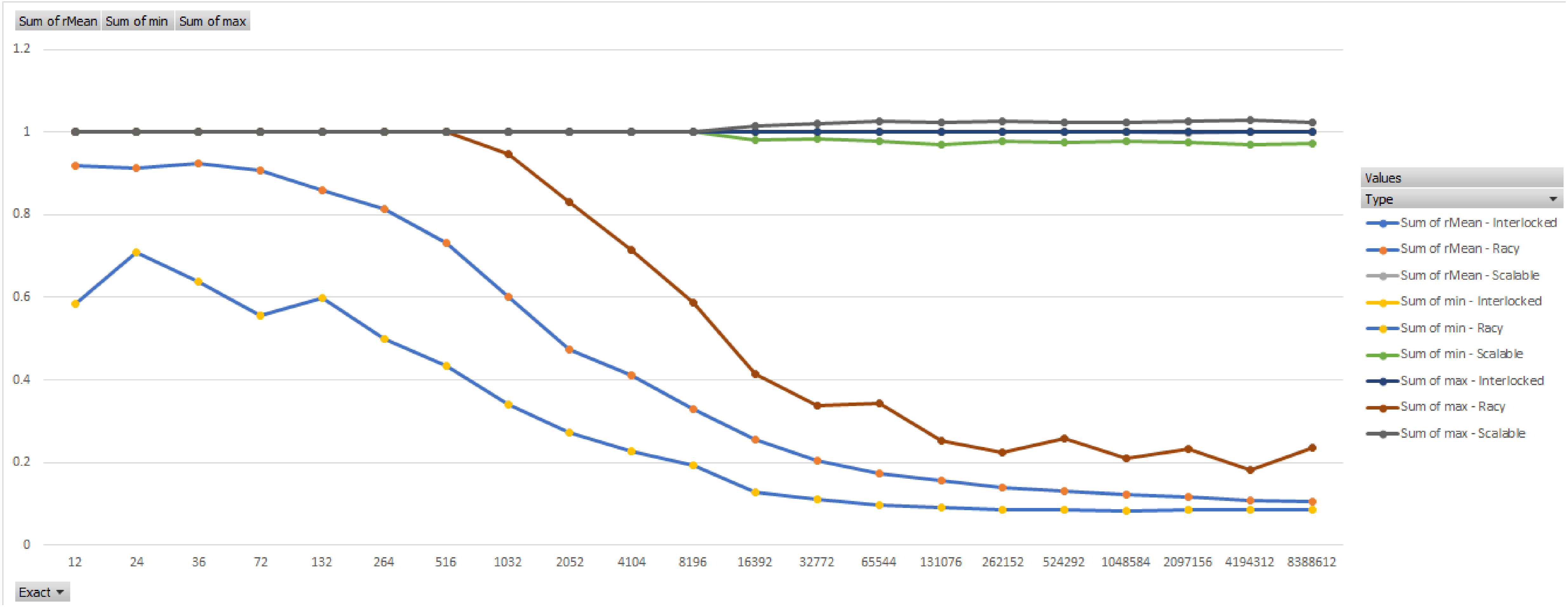
- Subset of edges get counters
    - Need to add in pseudo-edges
- Block counts reconstructed via “simple” math



Scalable Counters

- Pre .NET 8, instrumentation was using a shared counter for the sparse edge counts, and not interlocking ("racy") updates.
    - When app is heavily multithreaded:
        - Heavy contention on some counters (very slow Tier0-instr code)
        - Poor accuracy as many updates are lost due to races
    - Interlocked adds fix the accuracy issue, but contention is even worse
    - Not feasible to shard the counters (i.e., TLS) both because of the space required and the need to aggregate across shards

Scalable Counters

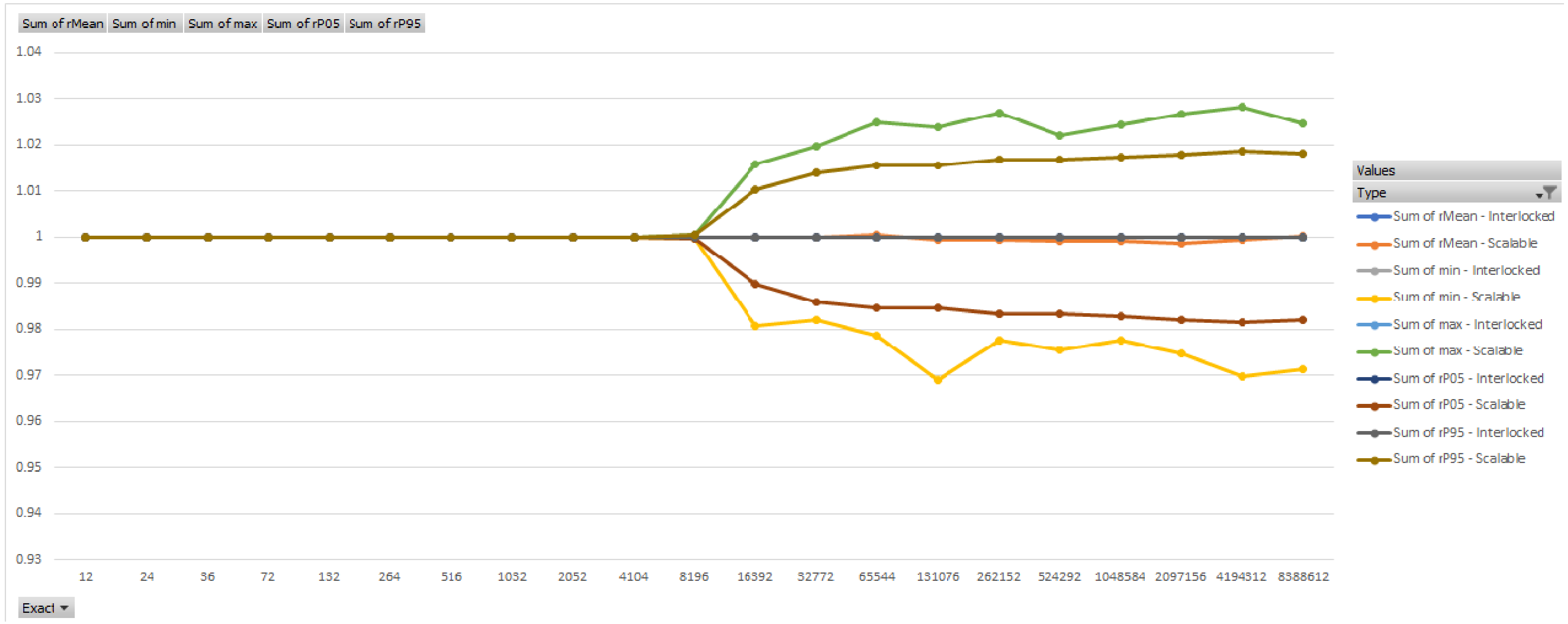


Blue and red lines show the ratio of a “racy” contended counter’s value to the true value. Note it can lose upwards of 90% of the counts (this was on a 12 core machine)

Scalable Counters

- .NET 8 introduces scalable counters
  - Use interlocked add for first 2^N counts
  - Add randomly after that…
    - Add by 2 with probability ½ for count in $(2^N, 2^{N+1}]$
    - Then by 4 with probability ¼ for count in $(2^{N+1}, 2^{N+2}]$
- With suitable threshold (N=13) count value is very likely within 2% of true value
- Number of writes to “hot” (potentially contended) counters drops dramatically

Scalable Counters

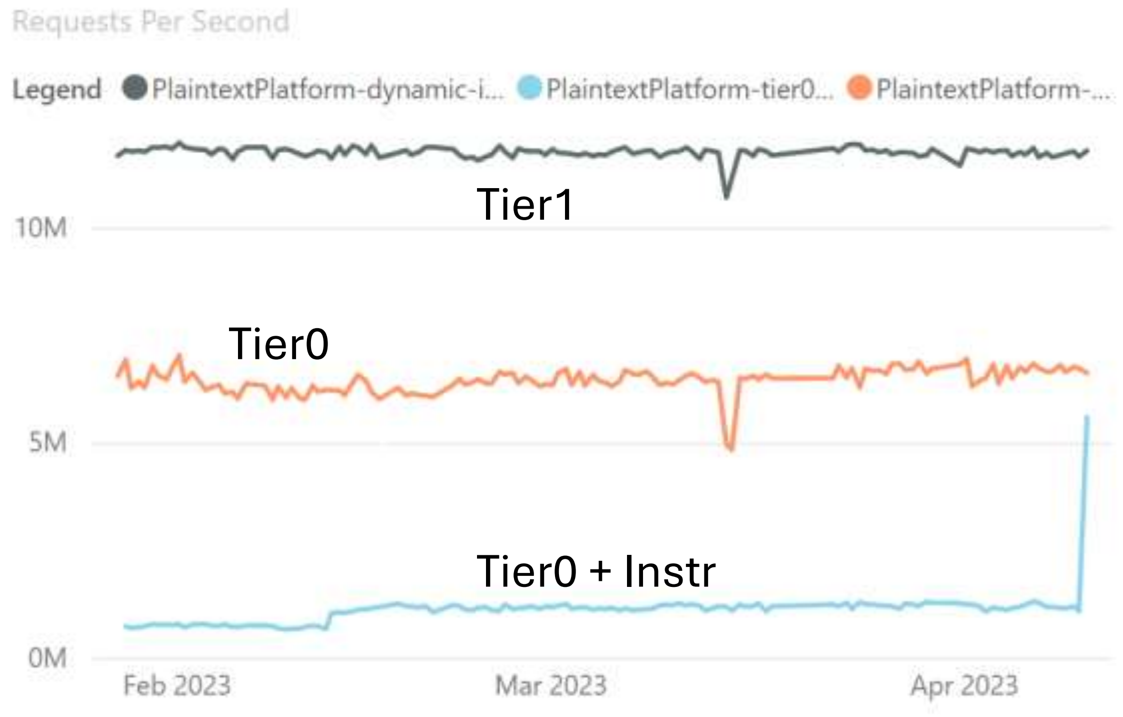


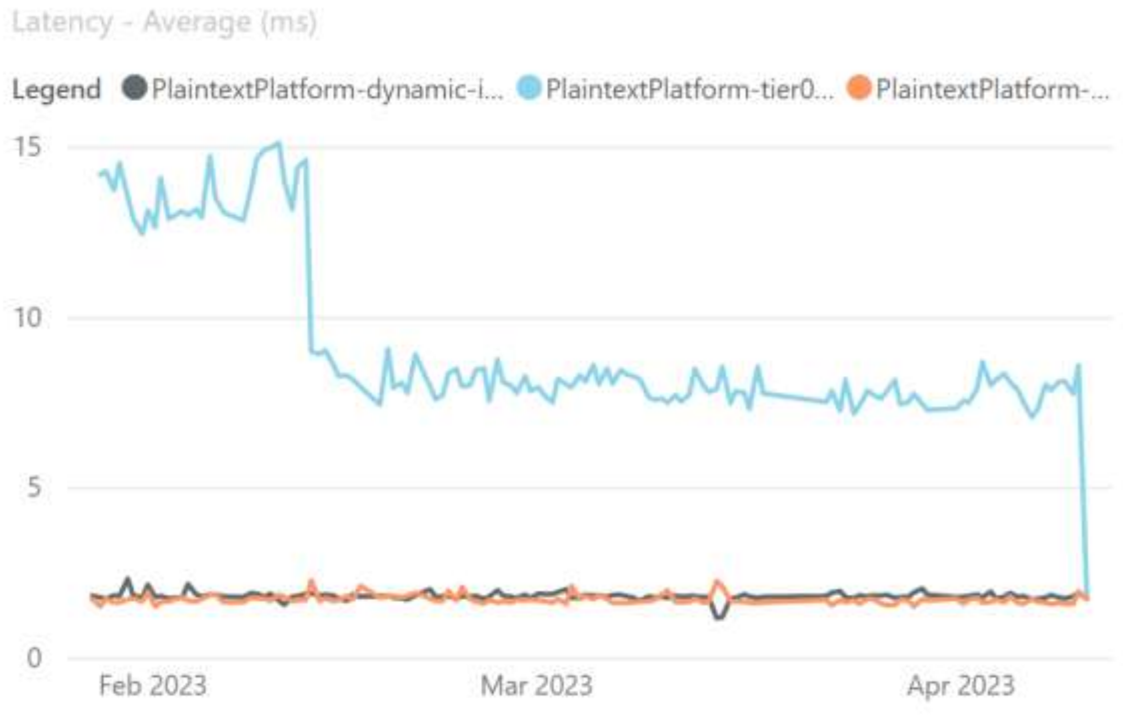
Deviation of scalable counter from true value.

Counts exactly up to 8192, then randomly for higher values

5-95 spread about +/- 2%

Impact of improvements to instrumentation on Tech Empower RPS / Latency

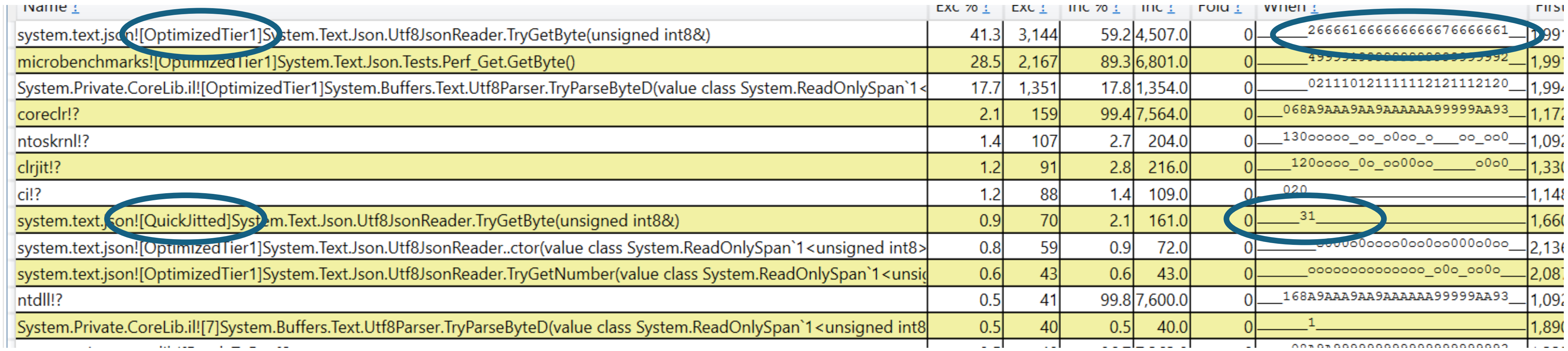




Randomness

- Instrumentation relies quite a bit on *randomness*
    - GDV profiles use randomness for Reservoir Sampling to build approximate histograms
    - Count profiles use randomness to improve scalability
- PGO data will likely not be the same from one run to the next
    - But typically, there are enough observations that the overall behavior is still stable and repeatable
    - There is already a fair amount of non-determinism when running code, but now the jitted codegen depends on it in a fundamental way.
- If you suspect a bug, try running with `DOTNET_TieredPGO=0`

PGO, BDN & PerfView



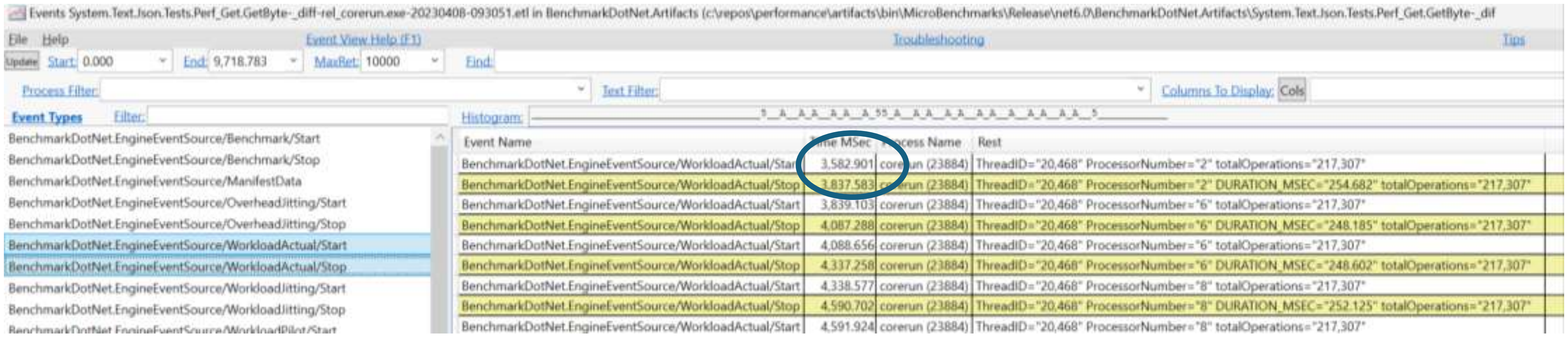
Name	Exc %	Exc	Inc %	Inc	Fold	When	Firs
system.text.json![OptimizedTier1]System.Text.Json.Utf8JsonReader.TryGetByte(unsigned int8&)	41.3	3,144	59.2	4,507.0	0	______2666616666666666676666661__	1,99
microbenchmarks![OptimizedTier1]System.Text.Json.Tests.Perf_Get.GetByte()	28.5	2,167	89.3	6,801.0	0	______4999919999999999999992__	1,99
System.Private.CoreLib.il![OptimizedTier1]System.Buffers.Text.Utf8Parser.TryParseByteD(value class System.ReadOnlySpan`1<	17.7	1,351	17.8	1,354.0	0	______0211101211111212111212 0__	1,99
coreclr!?	2.1	159	99.4	7,564.0	0	___068A9AAA9AA9AAAAAA99999AA93__	1,17
ntoskrnl!?	1.4	107	2.7	204.0	0	___130ooooo_oo_o0oo_o___oo_oo0__	1,09
clrjit!?	1.2	91	2.8	216.0	0	____120oooo_0o_oo0oo_____o0o0__	1,33
ci!?	1.2	88	1.4	109.0	0	___020_____________________	1,14
system.text.json![QuickJitted]System.Text.Json.Utf8JsonReader.TryGetByte(unsigned int8&)	0.9	70	2.1	161.0	0	_____31______________________	1,66
system.text.json![OptimizedTier1]System.Text.Json.Utf8JsonReader..ctor(value class System.ReadOnlySpan`1<unsigned int8>	0.8	59	0.9	72.0	0	_______ooooo0oooo0oo0oo000o0oo__	2,13
system.text.json![OptimizedTier1]System.Text.Json.Utf8JsonReader.TryGetNumber(value class System.ReadOnlySpan`1<unsig	0.6	43	0.6	43.0	0	______ooooooooooooo_o0o_oo0o___	2,08
ntdll!?	0.5	41	99.8	7,600.0	0	___168A9AAA9AA9AAAAAA99999AA93__	1,09
System.Private.CoreLib.il![7]System.Buffers.Text.Utf8Parser.TryParseByteD(value class System.ReadOnlySpan`1<unsigned int8	0.5	40	0.5	40.0	0	______1_______________________	1,89

You can get ETL traces from BDN via -p ETW (on windows) and view these with PerfView.

Note the same method now appears as two entries. From the time chart we can see that the QuickJitted (Tier0) version ran early on, and then the Tier1 version took over.

Make sure to filter the profile to just the time that BDN is actually making measurements.

PGO, BDN & PerfView



Events System.Text.Json.Tests.Perf_Get.GetByte-_diff-rel_corerun.exe-20230408-093051.etl in BenchmarkDotNet.Artifacts (c:\repos\performance\artifacts\bin\MicroBenchmarks\Release\net6.0\BenchmarkDotNet.Artifacts\System.Text.Json.Tests.Perf_Get.GetByte-_dif

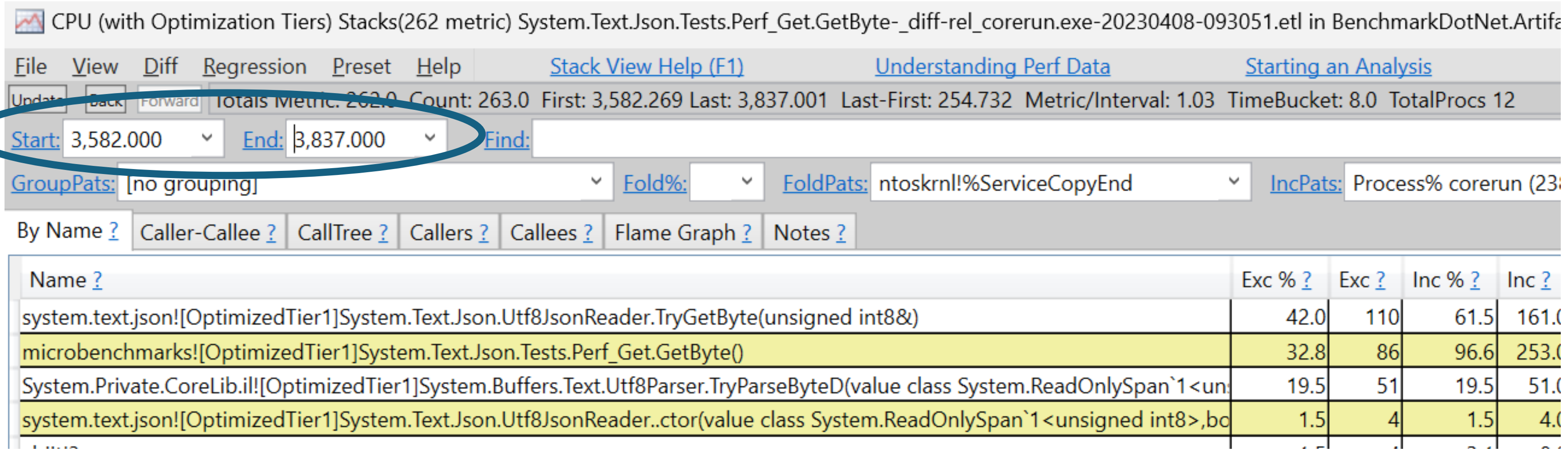
Start: 0.000 End: 9,718.783 MaxRet: 10000

Event Types

BenchmarkDotNet.EngineEventSource/Benchmark/Start
BenchmarkDotNet.EngineEventSource/Benchmark/Stop
BenchmarkDotNet.EngineEventSource/ManifestData
BenchmarkDotNet.EngineEventSource/OverheadJitting/Start
BenchmarkDotNet.EngineEventSource/OverheadJitting/Stop
BenchmarkDotNet.EngineEventSource/WorkloadActual/Start
BenchmarkDotNet.EngineEventSource/WorkloadActual/Stop
BenchmarkDotNet.EngineEventSource/WorkloadJitting/Start
BenchmarkDotNet.EngineEventSource/WorkloadJitting/Stop
BenchmarkDotNet.EngineEventSource/WorkloadPilot/Start

Event Name	Time MSec	Process Name	Rest
BenchmarkDotNet.EngineEventSource/WorkloadActual/Start	3,582.901	corerun (23884)	ThreadID="20,468" ProcessorNumber="2" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Stop	3,837.583	corerun (23884)	ThreadID="20,468" ProcessorNumber="2" DURATION_MSEC="254.682" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Start	3,839.103	corerun (23884)	ThreadID="20,468" ProcessorNumber="6" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Stop	4,087.288	corerun (23884)	ThreadID="20,468" ProcessorNumber="6" DURATION_MSEC="248.185" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Start	4,088.656	corerun (23884)	ThreadID="20,468" ProcessorNumber="6" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Stop	4,337.258	corerun (23884)	ThreadID="20,468" ProcessorNumber="6" DURATION_MSEC="248.602" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Start	4,338.577	corerun (23884)	ThreadID="20,468" ProcessorNumber="8" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Stop	4,590.702	corerun (23884)	ThreadID="20,468" ProcessorNumber="8" DURATION_MSEC="252.125" totalOperations="217,307"
BenchmarkDotNet.EngineEventSource/WorkloadActual/Start	4,591.924	corerun (23884)	ThreadID="20,468" ProcessorNumber="8" totalOperations="217,307"

Open the events view, select BD's WorkloadActual events, verify the intervals show consistent times, pick one and set the time limits on your profile view. Here: 3582..3837



CPU (with Optimization Tiers) Stacks(262 metric) System.Text.Json.Tests.Perf_Get.GetByte-_diff-rel_corerun.exe-20230408-093051.etl in BenchmarkDotNet.Artifa

Totals Metric: 262.0 Count: 263.0 First: 3,582.269 Last: 3,837.001 Last-First: 254.732 Metric/Interval: 1.03 TimeBucket: 8.0 TotalProcs 12

Start: 3,582.000 End: 3,837.000

GroupPats: [no grouping] Fold%: FoldPats: ntoskrnl!%ServiceCopyEnd IncPats: Process% corerun (23

Name ?	Exc % ?	Exc ?	Inc % ?	Inc ?
system.text.json![OptimizedTier1]System.Text.Json.Utf8JsonReader.TryGetByte(unsigned int8&)	42.0	110	61.5	161.(
microbenchmarks![OptimizedTier1]System.Text.Json.Tests.Perf_Get.GetByte()	32.8	86	96.6	253.(
System.Private.CoreLib.il![OptimizedTier1]System.Buffers.Text.Utf8Parser.TryParseByteD(value class System.ReadOnlySpan`1<un:	19.5	51	19.5	51.(
system.text.json![OptimizedTier1]System.Text.Json.Utf8JsonReader..ctor(value class System.ReadOnlySpan`1<unsigned int8>,bo	1.5	4	1.5	4.(