контур

Обзор перфомансных изменений в .NET 5

Рома Неволин

DevRel в Контуре

Launch of .NET 5

.NET 5 performance improvements

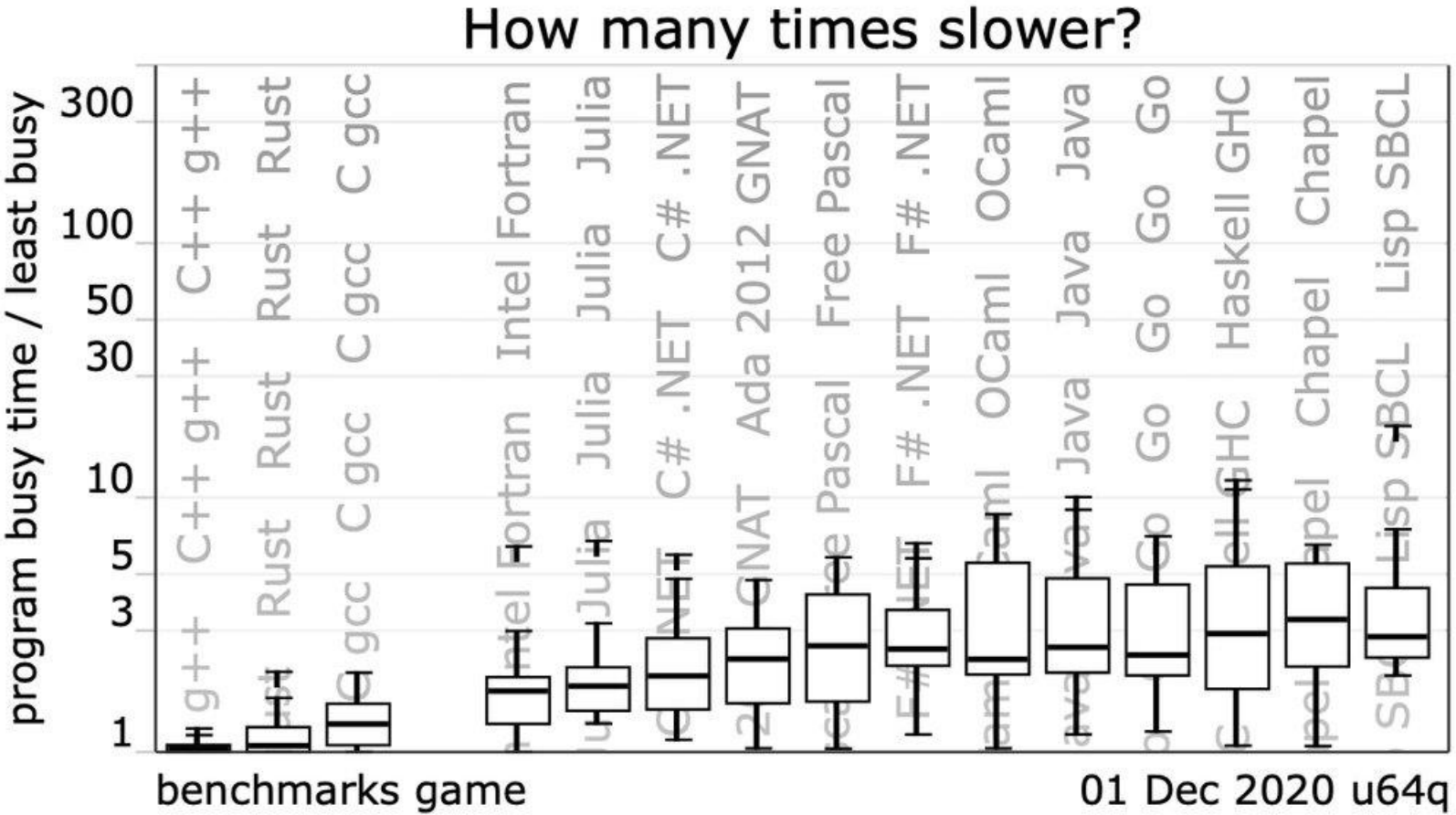
TechEmpower benchmarks round 20 predicted perf increases for .NET 5 over .NET Core 3.1

+38%
11,712,405 RPS
Plaintext

+42%
1,243,436 RPS
Json

+20%
417,818 RPS
Fortunes

Internal tests of .NET 5 performed on equivalent hardware as TechEmpower



benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html

Performance Improvements in .NET 5

Stephen

July 13th, 2020

In previous releases of .NET Core, I've blogged about the significant performance improvements that found their way into the release. For each post, from .NET Core 2.0 to .NET Core 2.1 to .NET Core 3.0, I found myself having more and more to talk about. Yet interestingly, after each I also found myself wondering whether there'd be enough meaningful improvements next time to warrant another post. Now that .NET 5 is shipping preview releases, I can definitively say the answer is, again, "yes". .NET 5 has already seen a wealth of performance improvements, and even though it's not scheduled for final release until later this year and there's very likely to be a lot more improvements that find their way in by then, I wanted to highlight a bunch of the improvements that are already available now. In this post, I'll highlight ~250 pull requests that have contributed to myriad of performance improvements across .NET 5.

Работа с текстом

```
[Benchmark]
public int Trim() => " test ".AsSpan().Trim().Length;
```

Method	Runtime	Mean	Ratio	Code Size
Trim	.NET FW 4.8	21.694 ns	1.00	569 B
Trim	.NET Core 3.1	8.079 ns	0.37	377 B
Trim	.NET 5.0	6.556 ns	0.30	365 B

Perf improvements to char.IsWhiteSpace / char.IsUpper / char.IsLower #26848

Open with

Merged GrabYourPitchfo... merged 3 commits into `dotnet:master` from `GrabYourPitchforks:char_updates` on 24 Sep 2019

Conversation 11 | Commits 3 | Checks 55 | Files changed 1 | +51 −86

GrabYourPitchforks commented on 24 Sep 2019 — Member

This is especially noticeable in benchmarks for APIs like `ReadOnlySpan.Trim()`.

Method	Toolchain	input	Mean	Error	StdDev	Median	Ratio	RatioSD
Trim	**master**	**{empty}**	**3.509 ns**	**0.0475 ns**	**0.0444 ns**	**3.501 ns**	**1.00**	**0.00**
Trim	proto	{empty}	3.423 ns	0.0801 ns	0.0710 ns	3.399 ns	0.98	0.03
Trim	**master**	**{spaces on both sides}**	**9.161 ns**	**0.2052 ns**	**0.2195 ns**	**9.091 ns**	**1.00**	**0.00**
Trim	proto	{spaces on both sides}	7.299 ns	0.1714 ns	0.1519 ns	7.287 ns	0.79	0.02

Reviewers

marek-safar

gfoidl

adamsitnik ✓

tarekgh ✓

Assignees

No one assigned

Labels

None yet

Projects

None yet

github.com/dotnet/coreclr/pull/26848

Improve char.ToUpperInvariant / ToLowerInvariant perf #35194

Open with

Merged GrabYourPitchfo... merged 1 commit into `dotnet:master` from `GrabYourPitchforks:invariant_perf` on 20 Apr 2020

Conversation 12 | Commits 1 | Checks 125 | Files changed 2 | +32 −17

GrabYourPitchforks commented on 20 Apr 2020 — Member

Somewhat inspired by Steve's PR at #35185. Turns out we call `char.ToUpperInvariant` / `char.ToLowerInvariant` in a few dozen places within the runtime.

This PR improves these code paths by: (a) not bouncing through the `TextInfo.Invariant` indirection for ASCII data or when `GlobalizationMode.Invariant` is enabled; (b) inlining small methods that the JIT wasn't already inlining; and (c) tweaking the bit twiddling to get more efficient codegen.

Method	Toolchain	args	Mean	Error	StdDev	Ratio	Code Size
CharToUpperInvariant	master	Hello World!	51.98 ns	0.615 ns	0.545 ns	1.00	147 B
CharToUpperInvariant	textinfo	Hello World!	22.71 ns	0.436 ns	0.408 ns	0.44	105 B
CharToLowerInvariant	master	Hello World!	49.15 ns	0.295 ns	0.247 ns	1.00	144 B
CharToLowerInvariant	textinfo	Hello World!	22.68 ns	0.242 ns	0.215 ns	0.46	105 B

Reviewers
- gfoidl
- stephentoub ✓
- tarekgh ✓

Assignees
No one assigned

Labels
area-System.Runtime, tenet-performance

Projects
None yet

github.com/dotnet/runtime/pull/35194

```
[Benchmark]
[Arguments("It's exciting to see great performance!")]
public int ToUpperInvariant(string s)
{
    int sum = 0;

    for (int i = 0; i < s.Length; i++)
        sum += char.ToUpperInvariant(s[i]);

    return sum;
}
```

Method	Runtime	Mean	Ratio	Code Size
ToUpperInvariant	.NET FW 4.8	208.34 ns	1.00	171 B
ToUpperInvariant	.NET Core 3.1	166.10 ns	0.80	164 B
ToUpperInvariant	.NET 5.0	69.15 ns	0.33	105 B

Optimize integral ToString (without generics) #32528

Merged **jkotas** merged 1 commit into `dotnet:master` from `ts2do:IntFormatNoGeneric` on 19 Feb 2020

Conversation 2 | Commits 1 | Checks 84 | Files changed 5

ts2do commented on 19 Feb 2020 — Contributor

- Make the fast path for Number.FormatXX & Number.TryFormatXX inlineable
- Make parameterless ToString in SByte, Int16, Int32, and Int64 directly invoke the default formatting method

This includes the changes from #32413 without the use of generics.
@jkotas: Can #32413 be used as the 'other' PR or should I remove the optimizations and open another?

 1  1

github.com/dotnet/runtime/pull/32528

```
[Benchmark]
public string Roundtrip()
{
    byte[] bytes = Encoding.UTF8.GetBytes("this is a test");
    return Encoding.UTF8.GetString(bytes);
}
```

Method	Runtime	Mean	Ratio	Allocated
Roundtrip	.NET FW 4.8	113.69 ns	1.00	96 B
Roundtrip	.NET Core 3.1	49.76 ns	0.44	96 B
Roundtrip	.NET 5.0	36.70 ns	0.32	96 B

Improve Encoding.UTF8.GetString / GetChars performance for small inputs #27268

Open with

Merged GrabYourPitchfo... merged 3 commits into `dotnet:master` from `GrabYourPitchforks:utf8_perf` on 9 Nov 2019

Conversation 18 | Commits 3 | Checks 52 | Files changed 3 | +120 −10

GrabYourPitchforks commented on 17 Oct 2019 — Member

This improves the performance of `Encoding.UTF8.GetString(byte[]) : string` and `Encoding.UTF8.GetBytes(string) : byte[]` by building on the existing JIT devirtualization logic and taking advantage of the case that most inputs to these functions are likely to be small (32 elements or fewer). For small inputs such as these, we already know that the maximum input size fits nicely into a stackalloc, so we can avoid the counting step and move straight to transcoding + the final memcpy.

Method	Toolchain	Text	Mean	Error	StdDev	Median	Ratio	Ra
GetString_FromByteArray	**master**		**1.098 ns**	**0.0619 ns**	**0.0549 ns**	**1.075 ns**	**1.00**	
GetString_FromByteArray	proto		8.682 ns	0.2368 ns	0.1978 ns	8.664 ns	7.91	
GetByteArray_FromString	master		21.109 ns	0.5399 ns	1.4032 ns	20.549 ns	1.00	
GetByteArray_FromString	proto		8.081 ns	0.1581 ns	0.1401 ns	8.025 ns	0.36	

Reviewers
- EgorBo
- gfoidl
- tarekgh ✓
- jkotas ✓

Assignees
No one assigned

Labels
area-System.Runtime

Projects
None yet

github.com/dotnet/coreclr/pull/27268

```
private readonly string _input = new HttpClient().GetStringAsync("http://www.gutenbe
private Regex _regex;

[Params(false, true)]
public bool Compiled { get; set; }

[GlobalSetup]
public void Setup() => _regex = new Regex(@"^.*\blove\b.*$", RegexOptions.Multiline

[Benchmark]
public int Count() => _regex.Matches(_input).Count;
```

Method	Runtime	Compiled	Mean	Ratio
Count	.NET FW 4.8	False	26.207 ms	1.00
Count	.NET Core 3.1	False	21.106 ms	0.80
Count	.NET 5.0	False	4.065 ms	0.16
Count	.NET FW 4.8	True	16.944 ms	1.00
Count	.NET Core 3.1	True	15.287 ms	0.90
Count	.NET 5.0	True	2.172 ms	0.13

Optimize newline handling for RegexOptions.Multiline #34566

Merged **stephentoub** merged 2 commits into `dotnet:master` from `stephentoub:bolffc` on 9 Apr 2020

Conversation 9 | Commits 2 | Checks 126 | Files changed 8

stephentoub commented on 5 Apr 2020 • edited ▾ Member ...

We previously didn't do any special handling of beginning-of-line anchors (`^` when `RegexOptions.Multiline` is specified). This PR adds special handling for the anchor so that `FindFirstChar` will jump to the next newline as part of its processing.

As part of this, I also cleaned up some of the anchor handling code. The `RegexPrefixAnalyzer` only ever returns a single anchor, but the rest of the code was written such that it was expecting multiple anchors.

Example:
Finding all lines in Romeo and Juliet that contain the word "love":

https://github.com/dotnet/runtime/pull/34566

Reduce unnecessary Regex match attempts for expressions beginning with atomic loops #35824

Open with

Merged **danmosemsft** merged 2 commits into `dotnet:master` from `stephentoub:atomicfast` on 5 May 2020

Conversation 7 | Commits 2 | Checks 116 | Files changed 9 | +137 −6

stephentoub commented on 5 May 2020 • edited — Member

Optimize Regex expressions that begin with atomic unbounded loops (either as written by the dev or because the system detected the loop could be made atomic and did it implicitly) by updating the starting position in the scan loop for the next iteration to be where the loop ended rather than where it started.

Running the examples from https://github.com/mariomka/regex-benchmark/blob/652d55810691ad88e1c2292a2646d301d3928903/csharp/Benchmark.cs#L20-L26:

Reviewers

 **danmosemsft** ✓

Assignees

No one assigned

github.com/dotnet/runtime/pull/35824

```
private Regex _email = new Regex(@"[\w\.+-]+@[\w\.-]+\.[\w\.-]+", RegexOptions.Compi
private Regex _uri = new Regex(@"[\w]+://[^/\s?#]+[^\s?#]+(?:\?[^\s#]*)?(?:#[^\s]*)?
private Regex _ip = new Regex(@"(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9])\.){3}(?:2

private string _input = new HttpClient().GetStringAsync("https://raw.githubuserconte

[Benchmark] public int Email() => _email.Matches(_input).Count;
[Benchmark] public int Uri() => _uri.Matches(_input).Count;
[Benchmark] public int IP() => _ip.Matches(_input).Count;
```

Method	Runtime	Mean	Ratio
Email	.NET FW 4.8	1,036.729 ms	1.00
Email	.NET Core 3.1	930.238 ms	0.90
Email	.NET 5.0	50.911 ms	0.05
Uri	.NET FW 4.8	870.114 ms	1.00
Uri	.NET Core 3.1	759.079 ms	0.87
Uri	.NET 5.0	50.022 ms	0.06
IP	.NET FW 4.8	75.718 ms	1.00
IP	.NET Core 3.1	61.818 ms	0.82
IP	.NET 5.0	6.837 ms	0.09

Коллекции

Improve performance of iterating with ForEach over ImmutableArray #1183

Open with

Merged **jkotas** merged 2 commits into `dotnet:master` from `hnrqbaggio:iterateforeach_immutablearray` on 27 Dec 2019

Conversation 4 | Commits 2 | Checks 71 | Files changed 1 | +2 −0

Changes from **all commits** | File filter... | Jump to... | 0 / 1 files viewed | Review changes

2 ...System.Collections.Immutable/src/System/Collections/Immutable/ImmutableArray_1.Minimal.cs Viewed

```
@@ -8,6 +8,7 @@
 8  8    using System.Diagnostics.Contracts;
 9  9    using System.Globalization;
10 10    using System.Linq;
   11 +  using System.Runtime.CompilerServices;
11 12    using System.Runtime.Versioning;
12 13
13 14    namespace System.Collections.Immutable
@@ -283,6 +284,7 @@ public ImmutableArray<T>.Builder ToBuilder()
283 284             /// </summary>
284 285             /// <returns>An enumerator.</returns>
285 286             [Pure]
   287 +           [MethodImpl(MethodImplOptions.AggressiveInlining)]
286 288             public Enumerator GetEnumerator()
287 289             {
288 290                 var self = this;
```

github.com/dotnet/runtime/pull/1183/files

```
private ImmutableArray<int> _array = ImmutableArray.Create(Enumerable.Range(0, 100_0

[Benchmark]
public int Sum()
{
    int sum = 0;

    foreach (int i in _array)
        sum += i;

    return sum;
}
```

Method	Runtime	Mean	Ratio
Sum	.NET FW 4.8	187.60 us	1.00
Sum	.NET Core 3.1	187.32 us	1.00
Sum	.NET 5.0	46.59 us	0.25

```
private ImmutableList<int> _list = ImmutableList.Create(Enumerable.Range(0, 1_000).T

[Benchmark]
public int Sum()
{
    int sum = 0;

    for (int i = 0; i < 1_000; i++)
        if (_list.Contains(i))
            sum += i;

    return sum;
}
```

Method	Runtime	Mean	Ratio
Sum	.NET FW 4.8	22.259 ms	1.00
Sum	.NET Core 3.1	22.872 ms	1.03
Sum	.NET 5.0	2.066 ms	0.09

```
[GlobalSetup]
public void Setup()
{
    var r = new Random(42);
    _array = Enumerable.Range(0, 1_000).Select(_ => r.Next()).ToArray();
}

private int[] _array;

[Benchmark]
public void Sort()
{
    foreach (int i in _array.OrderBy(i => i)) { }
}
```

Method	Runtime	Mean	Ratio
Sort	.NET FW 4.8	100.78 us	1.00
Sort	.NET Core 3.1	101.03 us	1.00
Sort	.NET 5.0	85.46 us	0.85

Avoid mod operator when fast alternative available #27299

Merged **jkotas** merged 12 commits into `dotnet:master` from `benaadams:dict-divisor` on 26 Oct 2019

Conversation 86 | Commits 12 | Checks 52 | Files changed 3

benaadams commented on 19 Oct 2019 • edited ▾ Member

Use fastmod when 128bit multiply is available. (by **@lemire** https://lemire.me/blog/2019/02/08/faster-remainders-when-the-divisor-is-a-constant-beating-compilers-and-libdivide/)

Api for review dotnet/corefx#41822 to enable 128bit multiply and make it more efficient.

Other half of PR from #27149 (first half was #27195)

github.com/dotnet/coreclr/pull/27299

```
int i = buckets[hashCode % (uint)buckets.Length];
int i = GetBucket(hashCode);
```

```
- int i = buckets[hashCode % (uint)buckets.Length];
+ int i = GetBucket(hashCode);
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private ref int GetBucket(uint hashCode)
{
    int[] buckets = _buckets!;
#if BIT64
    return ref buckets[HashHelpers.FastMod(hashCode, (uint)buckets.Length, _fastModMultiplier)];
#else
    return ref buckets[hashCode % (uint)buckets.Length];
#endif
}
```

Daniel Lemire's blog

Daniel Lemire is a computer science professor at the University of Quebec (TELUQ) in Montreal. His research is focused on software performance and data engineering. He is a techno-optimist.

My home page

My papers

My software

Faster remainders when the divisor is a constant: beating compilers and libdivide

Not all instructions on modern processors cost the same. Additions and subtractions are cheaper than multiplications which are themselves cheaper than divisions. For this reason, compilers frequently replace division instructions by multiplications. Roughly speaking, it works in this manner. Suppose that you want to divide a variable n by a constant d. You have that $n/d = n * (2^N/d) / (2^N)$. The division by a power of two ($/ (2^N)$) can be implemented as a right shift if we are working with unsigned integers, which compiles to single instruction: that is possible because the underlying hardware uses a base 2. Thus if $2^N/d$ has been precomputed, you can compute the division n/d as a multiplication and a shift. Of course, if d is not a power of two, $2^N/d$ cannot be represented as an integer. Yet for N large enough[footnote], we can approximate $2^N/d$ by an integer and have the exact computation of the remainder for all possible n within a range. I believe that all optimizing C/C++ compilers know how to pull this trick and it is generally beneficial irrespective of the processor's architecture.

The idea is not novel and goes back to at least 1973 (Jacobsohn). However, engineering matters because computer registers have finite number of bits, and multiplications can overflow. I believe that, historically, this

Granlund and Montgomery (1994). While GNU GCC and the Go compiler

lemire.me/blog/2019/02/08/faster-remainders-when-the-divisor-is-a-constant-beating-compilers-and-libdivide

```
private Dictionary<int, int> _dictionary = Enumerable.Range(0, 10_000).ToDictionary(

[Benchmark]
public int Sum()
{
    Dictionary<int, int> dictionary = _dictionary;
    int sum = 0;

    for (int i = 0; i < 10_000; i++)
        if (dictionary.TryGetValue(i, out int value))
            sum += value;

    return sum;
}
```

Method	Runtime	Mean	Ratio
Sum	.NET FW 4.8	77.45 us	1.00
Sum	.NET Core 3.1	67.35 us	0.87
Sum	.NET 5.0	44.10 us	0.57

Rewrite HashSet<T>'s implementation based on Dictionary<T>'s #37180

Open with

Merged **stephentoub** merged 9 commits into `dotnet:master` from `stephentoub:hashsetperf` on 1 Jun 2020

Conversation 52 | Commits 9 | Checks 186 | Files changed 15 | +2,003 −2,399

stephentoub commented on 30 May 2020 — Member

Fixes #37111
Contributes to #1989

This moves HashSet into corelib, and then effectively deletes HashSet's data structure and replaces it with the one used by Dictionary, then updated for the differences (e.g. just a value rather than a key and a value). HashSet used to have basically the same implementation, but Dictionary has evolved significantly and HashSet hasn't; this brings them to basic parity on implementation.

Based on perf tests, I veered away from Dictionary's implementation in a few places (e.g. a goto-based implementation in the core find method led to a significant regression for Int32-based Contains operations), and we should follow-up to understand whether Dictionary should be changed as well, or why there's a difference between the two. **@benaadams**, if you have some time, it'd probably worth looking at this again; maybe you'll get different numbers than I did.

Functionally, bringing over Dictionary's implementation yields a few notable changes, namely that Remove and Clear no longer invalidate enumerations. The tests have been updated accordingly.

With HashSet now in corelib, I also updated two Dictionary uses I found in corelib that were using Dictionary as a set and switched them to use HashSet.

Running the dotnet/performance perf tests:

```
dotnet run -c Release -f net5.0 --filter System.Collections.*.HashSet --corerun d:\coreclrtest\master\corerun.ex
```

Reviewers
- danmosemsft ✓
- jkotas
- AntonLapounov

Assignees
No one assigned

Labels
area-System.Collections
tenet-performance

Projects
None yet

Milestone
5.0.0

github.com/dotnet/runtime/pull/37180

New serializer converter model for objects and collections #2259

Open with

Merged **steveharter** merged 3 commits into `dotnet:master` from `steveharter:NewConverterModel` on 12 Feb 2020

Conversation 507 | Commits 3 | Checks 66 | Files changed 108 | +6,049 −4,906

steveharter commented on 28 Jan 2020 • edited by richlander Member

As part of #1562 this PR refactors the converter model for objects and collection.

No public APIs have changed with this PR - changes were made to be internal for now.

Short-term this PR improves perf and maintainability.

Longer-term a subsequent PR will be created for the proposed public API changes which are expected to add capabilities as explained in the above link (mostly supporting code-gen for AOT and adding public APIs for extensibility and new converter capabilities).

Reviewers

 layomia

 Jozkee

 safern

 ahsonkhan

Assignees

 steveharter

github.com/dotnet/runtime/pull/2259

```
private MemoryStream _stream = new MemoryStream();
private DateTime[] _array = Enumerable.Range(0, 1000).Select(_ => DateTime.UtcNow).T

[Benchmark]
public Task LargeArray()
{
    _stream.Position = 0;
    return JsonSerializer.SerializeAsync(_stream, _array);
}
```

Method	Runtime	Mean	Ratio	Allocated
LargeArray	.NET FW 4.8	262.06 us	1.00	24256 B
LargeArray	.NET Core 3.1	191.34 us	0.73	24184 B
LargeArray	.NET 5.0	69.40 us	0.26	152 B

Сеть

```
[Benchmark]
public Uri Ctor() => new Uri("https://github.com/dotnet/runtime/pull/36915");
```

Method	Runtime	Mean	Ratio	Allocated
Ctor	.NET FW 4.8	443.2 ns	1.00	225 B
Ctor	.NET Core 3.1	192.3 ns	0.43	72 B
Ctor	.NET 5.0	129.9 ns	0.29	56 B

```
private Uri _uri = new Uri("http://github.com/dotnet/runtime");

[Benchmark]
public string PathAndQuery() => _uri.PathAndQuery;
```

Method	Runtime	Mean	Ratio	Allocated
PathAndQuery	.NET FW 4.8	17.936 ns	1.00	56 B
PathAndQuery	.NET Core 3.1	30.891 ns	1.72	56 B
PathAndQuery	.NET 5.0	2.854 ns	0.16	–

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Hosting;

public class Program
{
    public static void Main(string[] args) =>
        Host.CreateDefaultBuilder(args).ConfigureWebHostDefaults(b => b.UseStartup<S
}

public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", context => context.Response.WriteAsync("Hello"));
            endpoints.MapPost("/", context => context.Response.WriteAsync("Hello"));
        });
    }
}
```

```
private HttpMessageInvoker _client = new HttpMessageInvoker(new SocketsHttpHandler()
private HttpRequestMessage _get = new HttpRequestMessage(HttpMethod.Get, new Uri("ht
private HttpRequestMessage _post = new HttpRequestMessage(HttpMethod.Post, new Uri("

[Benchmark] public Task Get() => MakeRequest(_get);

[Benchmark] public Task Post() => MakeRequest(_post);

private Task MakeRequest(HttpRequestMessage request) => Task.WhenAll(Enumerable.Rang
{
    for (int i = 0; i < 500; i++)
    {
        using (HttpResponseMessage r = await _client.SendAsync(request, default))
        using (Stream s = await r.Content.ReadAsStreamAsync())
            await s.CopyToAsync(Stream.Null);
    }
}));
```

Method	Runtime	Mean	Ratio	Allocated
Get	.NET Core 3.1	1,267.4 ms	1.00	122.76 MB
Get	.NET 5.0	681.7 ms	0.54	74.01 MB
Post	.NET Core 3.1	1,464.7 ms	1.00	280.51 MB
Post	.NET 5.0	735.6 ms	0.50	132.52 MB

Потоки и асинхронность

```
[Benchmark]
public async Task ValueTaskCost()
{
    for (int i = 0; i < 1_000; i++)
        await YieldOnce();
}

private static async ValueTask YieldOnce() => await Task.Yield();
```

Method	Runtime	Mean	Ratio	Allocated
ValueTaskCost	.NET FW 4.8	1,635.6 us	1.00	294010 B
ValueTaskCost	.NET Core 3.1	842.7 us	0.51	120184 B
ValueTaskCost	.NET 5.0	812.3 us	0.50	186 B

Make "async ValueTask/ValueTask<T>" methods ammortized allocation-free #26310

Open with

Merged **stephentoub** merged 1 commit into `dotnet:master` from `stephentoub:asyncvaluetaskallocs` on 24 Oct 2019

Conversation 84 | Commits 1 | Checks 52 | Files changed 7 | +654 −139

stephentoub commented on 22 Aug 2019 • edited — Member

Today `async ValueTask/ValueTask<T>` methods use builders that special-case the synchronously completing case (to just return a `default(ValueTask)` or `new ValueTask<T>(result)`) but defer to the equivalent of `async Task/Task<T>` for when the operation completes asynchronously. This, however, doesn't take advantage of the fact that value tasks can wrap arbitrary `IValueTaskSource/IValueTaskSource<T>` implementations.

I've had this sitting on the shelf for a while, but finally cleaned it up. The first three commits here are just moving around existing code. The last two commits are the meat of this change. This changes `AsyncValueTaskMethodBuilder` and `AsyncValueTaskMethodBuilder<T>` to use pooled `IValueTaskSource/IValueTaskSource<T>` instances, such that calls to an `async ValueTask/ValueTask<T>` method incur 0 allocations as long as there's a cached object available.

I've marked this as a draft and work-in-progress for a few reasons:

1. There's a breaking change here, in that while we say/document that `ValueTask/ValueTask<T>`s are more limited in what they can be used for, nothing in the implementation actually stops a `ValueTask` that was wrapping a `Task` from being used as permissively as `Task`, e.g. if you were to await such a `ValueTask` multiple times, it would happen to work, even though we say "never do that". This change means that anyone who was relying on such undocumented behaviors will now be broken. I think this is a reasonable thing to do in a major release, but I also want feedback and a lot of runway on it. There's the potential to make it opt-in (or opt-out) as well, but that will also non-trivially complicate the implementation.
2. Policy around pooling. This is always a difficult thing to tune. Right now I've chosen a policy that limits the number of pooled objects per state machine type to an arbitrary multiple of the processor count, and that tries to strike a balance between contention and garbage by using a spin lock and if there's any contention while trying to get or return a pooled object, the cache is ignored. We will need to think hard about what policy to use here. It's also possible it could be tuned per state machine, e.g. by having an attribute that's looked up via reflection when creating the cache for a state machine, but that'll add a lot of first-access overhead to any `async ValueTask/ValueTask<T>` method.
3. Perf validation.

cc: **@kouvel**, **@tarekgh**, **@benaadams**

👍 30 ❤️ 14

Reviewers
- davidfowl
- benaadams

Assignees
No one assigned

Labels
area-System.Threading

Projects
None yet

Milestone
5.0

Linked issues
Successfully merging this pull request may close these issues.
None yet

Notifications — Customize
Subscribe
You're not receiving notifications from this thread.

github.com/dotnet/coreclr/pull/26310

Async ValueTask Pooling in .NET 5

Stephen

March 16th, 2020

The async/await feature in C# has revolutionized how developers targeting .NET write asynchronous code. Sprinkle some `async` and `await` around, change some return types to be tasks, and badda bing badda boom, you've got an asynchronous implementation. In theory.

In practice, obviously I've exaggerated the ease with which a codebase can be made fully asynchronous, and as with many a software development task, the devil is often in the details. One such "devil" that performance-minded .NET developers are likely familiar with is the state machine object that enables an async method to perform its magic.

Improve ContinueWith perf with NotOn* options #35575

Open with

Merged **stephentoub** merged 5 commits into `dotnet:master` from `stephentoub:cwperf` on 2 May 2020

Conversation 2 | Commits 5 | Checks 129 | Files changed 2 | +87 −95

stephentoub commented on 28 Apr 2020 — Member

This came about while looking at allocations and CPU costs in some HttpClient code. When ContinueWith is used with TaskContinuationOptions.NotOn* options, when the antecedent task completes we compare the state of that antecedent task against the NotOn* options: if the options permit it, the continuation is queued/invoked, and if they don't, the continuation is canceled. That cancellation then ends up being common in cases where a ContinueWith is used, for example, to log exceptions that result from a faulted task, e.g.

```
private static void Log(Task task) =>
    task.ContinueWith(t => Log(t.Exception),
        CancellationToken.None,
        TaskContinuationOptions.OnlyOnFaulted | TaskContinuationOptions.ExecuteSynchronously,
        TaskScheduler.Default);
```

We can handle that cancellation much more efficiently than we are today. Today that's resulting in us expanding the ContingentProperties inside of the task, in order to store that cancellation was requested internally, but that's not actually necessary. It's also resulting in us doing several atomic transitions via interlockeds, but that's only necessary if the task could have transitioned in any way, which is only possible if a cancelable token is provided.

This PR removes that overhead. It also shrinks the size of the object created by ContinueWith by a field, seals and renames it, and removes some dead code from related code paths.

Reviewers

tarekgh ✓

Assignees

No one assigned

Labels

area-System.Threading.Tasks

tenet-performance

Projects

None yet

Milestone

5.0.0

github.com/dotnet/runtime/pull/35575

```
const int Iters = 1_000_000;

private AsyncTaskMethodBuilder[] tasks = new AsyncTaskMethodBuilder[Iters];

[IterationSetup]
public void Setup()
{
    Array.Clear(tasks, 0, tasks.Length);
    for (int i = 0; i < tasks.Length; i++)
        _ = tasks[i].Task;
}

[Benchmark(OperationsPerInvoke = Iters)]
public void Cancel()
{
    for (int i = 0; i < tasks.Length; i++)
    {
        tasks[i].Task.ContinueWith(_ => { }, CancellationToken.None, TaskContinuatio
        tasks[i].SetResult();
    }
}
```

Method	Runtime	Mean	Ratio	Allocated
Cancel	.NET FW 4.8	239.2 ns	1.00	193 B
Cancel	.NET Core 3.1	140.3 ns	0.59	192 B
Cancel	.NET 5.0	106.4 ns	0.44	112 B

JIT

```
[Benchmark]
public int Zeroing()
{
    ReadOnlySpan<char> s1 = "hello world";
    ReadOnlySpan<char> s2 = Nop(s1);
    ReadOnlySpan<char> s3 = Nop(s2);
    ReadOnlySpan<char> s4 = Nop(s3);
    ReadOnlySpan<char> s5 = Nop(s4);
    ReadOnlySpan<char> s6 = Nop(s5);
    ReadOnlySpan<char> s7 = Nop(s6);
    ReadOnlySpan<char> s8 = Nop(s7);
    ReadOnlySpan<char> s9 = Nop(s8);
    ReadOnlySpan<char> s10 = Nop(s9);
    return s1.Length + s2.Length + s3.Length + s4.Length + s5.Length + s6.Length + s
}

[MethodImpl(MethodImplOptions.NoInlining)]
private static ReadOnlySpan<char> Nop(ReadOnlySpan<char> span) => default;
```

Method	Runtime	Mean	Ratio
Zeroing	.NET FW 4.8	22.85 ns	1.00
Zeroing	.NET Core 3.1	18.60 ns	0.81
Zeroing	.NET 5.0	15.07 ns	0.66

Use xmm for stack prolog zeroing rather than rep stos #32538

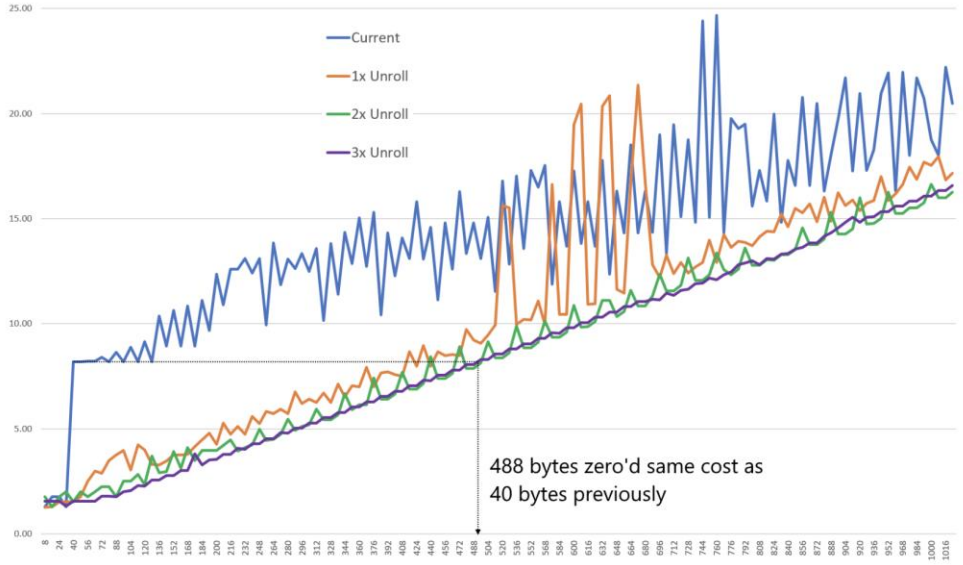
Merged **AndyAyersMS** merged 17 commits into `dotnet:master` from `benaadams:zero-init-xmm` on 5 Mar 2020

Conversation 217 · Commits 17 · Checks 149 · Files changed 8

benaadams commented on 19 Feb 2020 • edited — Member

Use `xmm` registers to clear prolog blocks rather than `rep stosd`.

`rep stos` does have a shallower growth rate so it will eventually overtake; however unlikely in the size range for the stack clearance:



Also its variability based on small changes in stack changes is problematic as just adding an extra variable to a method (with no other changes) could add or remove 9ns from its running time. Whereas the xmm clear in this PR is much smoother and more expected c[illegible]

github.com/dotnet/runtime/pull/32538

Optimization to remove redundant zero initializations. #36918

Merged **erozenfeld** merged 2 commits into `dotnet:master` from `erozenfeld:ZeroInitDiff` on 27 May 2020

Conversation 31 | Commits 2 | Checks 177 | Files changed 10

erozenfeld commented on 23 May 2020 — Member

This change adds a phase that iterates over basic blocks starting with the first basic block until there is no unique basic block successor or until it detects a loop. It keeps track of local nodes it encounters. When it gets to an assignment to a local variable or a local field, it checks whether the assignment is the first reference to the local (or to the parent of the local field), and, if so, it may do one of two optimizations:

1. If the local is untracked, the rhs of the assignment is 0, and the local is guaranteed to be fully initialized in the prolog, the explicit zero initialization is removed.
2. If the assignment is to a local (and not a field) and either the local has no gc pointers or there are no gc-safe points between the prolog and the assignment, it marks the local with lvHasExplicitInit which tells the codegen not to insert zero initialization for this local in the prolog.

This addresses one of the examples in #2325 and 5 examples in #1007.

👍 13 🚀 5

```
public static char Get(string s, int i) => s[i];
```

```
; Program.Get(System.String, Int32)
       sub       rsp,28
       cmp       edx,[rcx+8]
       jae       short M01_L00
       movsxd    rax,edx
       movzx     eax,word ptr [rcx+rax*2+0C]
       add       rsp,28
       ret
M01_L00:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 28
```

```
public static char Get(string s, int i) => s[i];
```

```
; Program.Get(System.String, Int32)
       sub       rsp,28
       cmp       edx,[rcx+8]
       jae       short M01_L00
       movsxd    rax,edx
       movzx     eax,word ptr [rcx+rax*2+0C]
       add       rsp,28
       ret
M01_L00:
       call      CORINFO_HELP_RNGCHKFAIL
       int       3
; Total bytes of code 28
```

```
private (int i, int j) _value;

[Benchmark]
public int NullCheck() => _value.j++;
```

```
; Program.NullCheck()
       nop       dword ptr [rax+rax]
       cmp       [rcx],ecx
       add       rcx,8
       add       rcx,4
       mov       eax,[rcx]
       lea       edx,[rax+1]
       mov       [rcx],edx
       ret
; Total bytes of code 23
```

```
private (int i, int j) _value;

[Benchmark]
public int NullCheck() => _value.j++;
```

```
; Program.NullCheck()
       nop         dword ptr [rax+rax]
       cmp         [rcx],ecx
       add         rcx,8
       add         rcx,4
       mov         eax,[rcx]
       lea         edx,[rax+1]
       mov         [rcx],edx
       ret
; Total bytes of code 23
```

```
private (int i, int j) _value;

[Benchmark]
public int NullCheck() => _value.j++;
```

```
; Program.NullCheck()
       nop         dword ptr [rax+rax]
       cmp         [rcx],ecx
       add         rcx,8
       add         rcx,4
       mov         eax,[rcx]
       lea         edx,[rax+1]
       mov         [rcx],edx
       ret
; Total bytes of code 23
```

```
private (int i, int j) _value;

[Benchmark]
public int NullCheck() => _value.j++;
```

```
; Program.NullCheck()
       add        rcx,0C
       mov        eax,[rcx]
       lea        edx,[rax+1]
       mov        [rcx],edx
       ret
; Total bytes of code 12
```

```
private int[] _array = Enumerable.Range(0, 1000).ToArray();

[Benchmark]
public bool IsSorted() => IsSorted(_array);

private static bool IsSorted(ReadOnlySpan<int> span)
{
    for (int i = 0; i < span.Length - 1; i++)
        if (span[i] > span[i + 1])
            return false;

    return true;
}
```

```
private int[] _array = Enumerable.Range(0, 1000).ToArray();

[Benchmark]
public bool IsSorted() => IsSorted(_array);

private static bool IsSorted(ReadOnlySpan<int> span)
{
    for (int i = 0; i < span.Length - 1; i++)
        if (span[i] > span[i + 1])
            return false;

    return true;
}
```

Method	Runtime	Mean	Ratio	Code Size
IsSorted	.NET FW 4.8	1,083.8 ns	1.00	236 B
IsSorted	.NET Core 3.1	581.2 ns	0.54	136 B
IsSorted	.NET 5.0	463.0 ns	0.43	105 B

Рома Неволин

 **nevoroman@gmail.com**

 **t.me/nevoroman**