Our way to TechEmpower wins in .NET 5

Adam Sitnik

.NET 5

Scenario: .NET 5 has excellent fundamentals

In our scope of fundamentals, we are including reliability, performance, diagnosability, compliance, security, acquisition and deployment. We will continue to deliver across these areas in .NET 5. The following list does not cover all of the things we intend to get done, but highlight some key deliverables in this space

1. Attain top 10 status for the Fortunes benchmark, and outcompete Netty (Java) on JSON serialization

https://devdiv.visualstudio.com/DevDiv/_wiki/wikis/DevDiv.wiki/6748/NET-5-Framing-Memo

Benchmarks Specifications

<u>JSON Serialization</u>: Exercises the framework fundamentals including keep-alive support, request routing, request header parsing, object instantiation, JSON serialization, response header generation, and request count throughput.

Single Database Query: Exercises the framework's object-relational mapper (ORM), random number generator, database driver, and database connection pool.

Multiple Database Queries: A variation of Test #2 and also uses the **World** table. Multiple rows are fetched to more dramatically punish the database driver and connection pool. At the highest queries-per-request tested (20), this test demonstrates all frameworks' convergence toward zero requests-per-second as database activity increases.

Fortunes: Exercises the ORM, database connectivity, dynamic-size collections, sorting, server-side templates, XSS countermeasures, and character encoding.

Database Updates: A variation of Test #3 that exercises the ORM's persistence of objects and the database driver's performance at running UPDATE statements or similar. The spirit of this test is to exercise a variable number of read-then-write style database operations.

Plaintext: An exercise of the request-routing fundamentals only, designed to demonstrate the capacity of highperformance platforms in particular. Requests will be sent using HTTP pipelining. The response payload is still small, meaning good performance is still necessary in order to saturate the gigabit Ethernet of the test environment.

Caching: Exercises the platform or framework's in-memory caching of information sourced from a database. For implementation simplicity, the requirements are very similar to the multiple database query test (Test #3), but use a separate database table and are fairly generous/forgiving, allowing for each platform or framework's best practices to be applied.

https://github.com/TechEmpower/FrameworkBenchmarks/wiki/Project-Information-Framework-Tests-Overview#test-types

Plaintext

Middleware

public class PlaintextMiddleware

private static readonly PathString _path = new PathString(Scenarios.GetPath(s => s.Plaintext));
private static readonly byte[] _helloWorldPayload = Encoding.UTF8.GetBytes("Hello, World!");

private readonly RequestDelegate _next;

public PlaintextMiddleware(RequestDelegate next) => _next = next;

public Task Invoke(HttpContext httpContext)

if (httpContext.Request.Path.StartsWithSegments(_path, StringComparison.Ordinal))

return WriteResponse(httpContext.Response);

return _next(httpContext);

public static Task WriteResponse(HttpResponse response)

```
var payloadLength = _helloWorldPayload.Length;
response.StatusCode = 200;
response.ContentType = "text/plain";
response.ContentLength = payloadLength;
return response.Body.WriteAsync(_helloWorldPayload, 0, payloadLength);
```

public static class PlaintextMiddlewareExtensions

public static IApplicationBuilder UsePlainText(this IApplicationBuilder builder)

return builder.UseMiddleware<PlaintextMiddleware>();

Platform

private readonly static AsciiString _plainTextBody = "Hello, World!";

private readonly static AsciiString _plaintextPreamble =

- _http110K +
 headerServer + crlf +
- headerContentTypeText + crlf +
- ______headerContentLength + _plainTextBody.Length.ToString();

private static void PlainText(ref BufferWriter<WriterAdapter> writer)

writer.Write(_plaintextPreamble);

// Date header
writer.Write(DateHeader.HeaderBytes);

// Body
writer.Write(_plainTextBody);

JSON

```
private readonly static uint _jsonPayloadSize
```

= (uint)JsonSerializer.SerializeToUtf8Bytes(new JsonMessage { message = "Hello, World!" }, SerializerOptions).Length;

```
private readonly static AsciiString _jsonPreamble =
```

```
_http110K +
_headerServer + _crlf +
_headerContentTypeJson + _crlf +
headerContentLength + jsonPayloadSize.ToString();
```

private static void Json(ref BufferWriter<WriterAdapter> writer, IBufferWriter<byte> bodyWriter)

```
writer.Write(_jsonPreamble);
// Date header
writer.Write(DateHeader.HeaderBytes);
writer.Commit();
```

```
Utf8JsonWriter utf8JsonWriter = t_writer ??= new Utf8JsonWriter(bodyWriter, new JsonWriterOptions { SkipValidation = true });
utf8JsonWriter.Reset(bodyWriter);
```

```
// Body
```

JsonSerializer.Serialize<JsonMessage>(utf8JsonWriter, new JsonMessage { message = "Hello, World!" }, SerializerOptions);

Single Query

var cmd = new NpgsqlCommand("SELECT id, randomnumber FROM world WHERE id = @Id", connection); var parameter = new NpgsqlParameter<int>(parameterName: "@Id", value: _random.Next(1, 10001));

private async Task SingleQuery(PipeWriter pipeWriter) => OutputSingleQuery(pipeWriter, await Db.LoadSingleQueryRow());

```
private static void OutputSingleQuery(PipeWriter pipeWriter, World row)
   var writer = GetWriter(pipeWriter, sizeHint: 180); // in reality it's 150
   writer.Write( dbPreamble);
   var lengthWriter = writer;
   writer.Write( contentLengthGap);
   // Date header
   writer.Write(DateHeader.HeaderBytes);
   writer.Commit();
   Utf8JsonWriter utf8JsonWriter = t writer ??= new Utf8JsonWriter(pipeWriter, new JsonWriterOptions { SkipValidation = true });
   utf8JsonWriter.Reset(pipeWriter);
   // Body
   JsonSerializer.Serialize<World>(utf8JsonWriter, row, SerializerOptions);
   // Content-Length
   lengthWriter.WriteNumeric((uint)utf8JsonWriter.BytesCommitted);
```

Multiple Queries

```
public async Task<World[]> LoadMultipleQueriesRows(int count)
    var result = new World[count];
    using (var db = new NpgsqlConnection( connectionString))
        await db.OpenAsync();
        var (cmd, idParameter) = CreateReadCommand(db);
        using (cmd)
            for (int i = 0; i < result.Length; i++)</pre>
                result[i] = await ReadSingleRow(cmd);
                idParameter.TypedValue = random.Next(1, 10001);
    return result;
```

private async Task MultipleQueries(PipeWriter pipeWriter, int count)
 => OutputMultipleQueries(pipeWriter, await Db.LoadMultipleQueriesRows(count));

private static void OutputMultipleQueries(PipeWriter pipeWriter, World[] rows)

var writer = GetWriter(pipeWriter, sizeHint: 160 * rows.Length); // in reality it's 152 for one

writer.Write(_dbPreamble);

var lengthWriter = writer; writer.Write(_contentLengthGap);

// Date header
writer.Write(DateHeader.HeaderBytes);

writer.Commit();

Utf8JsonWriter utf8JsonWriter = t_writer ??= new Utf8JsonWriter(pipeWriter, new JsonWriterOptions
utf8JsonWriter.Reset(pipeWriter);

```
// Body
JsonSerializer.Serialize<World[]>(utf8JsonWriter, rows, SerializerOptions);
```

// Content-Length
lengthWriter.WriteNumeric((uint)utf8JsonWriter.BytesCommitted);

Caching

```
private readonly Microsoft.Extensions.Caching.Memory.MemoryCache _cache = new MemoryCache(
    new MemoryCacheOptions()
    {
        ExpirationScanFrequency = TimeSpan.FromMinutes(60)
    });
public Task<World[]> LoadCachedQueries(int count)
```

```
var result = new World[count];
var cacheKeys = _cacheKeys;
var cache = _cache;
var random = _random;
for (var i = 0; i < result.Length; i++)
{
    var id = random.Next(1, 10001);
    var key = cacheKeys[id];
    var data = cache.Get<CachedWorld>(key);
    if (data != null)
    {
        result[i] = data;
    }
```

else

return LoadUncachedQueries(id, i, count, this, result);

private async Task Caching(PipeWriter pipeWriter, int count)

OutputMultipleQueries(pipeWriter, await Db.LoadCachedQueries(count));

Updates

```
public async Task<World[]> LoadMultipleUpdatesRows(int count)
   var results = new World[count];
   using (var db = new NpgsqlConnection(_connectionString))
                                                                                                                private async Task Updates(PipeWriter pipeWriter, int count)
                                                                                                                    => OutputUpdates(pipeWriter, await Db.LoadMultipleUpdatesRows(count));
       await db.OpenAsync();
                                                                                                                private static void OutputUpdates(PipeWriter pipeWriter, World[] rows)
       var (queryCmd, queryParameter) = CreateReadCommand(db);
       using (queryCmd)
                                                                                                                    var writer = GetWriter(pipeWriter, sizeHint: 120 * rows.Length); // in reality it's 112 for one
           for (int i = 0; i < results.Length; i++)</pre>
                                                                                                                    writer.Write(_dbPreamble);
              results[i] = await ReadSingleRow(queryCmd);
              queryParameter.TypedValue = _random.Next(1, 10001);
                                                                                                                    var lengthWriter = writer;
                                                                                                                    writer.Write( contentLengthGap);
       using (var updateCmd = new NpgsqlCommand(BatchUpdateString.Query(count), db))
                                                                                                                    // Date header
                                                                                                                    writer.Write(DateHeader.HeaderBytes);
           var ids = BatchUpdateString.Ids;
           var randoms = BatchUpdateString.Randoms;
                                                                                                                    writer.Commit();
           for (int i = 0; i < results.Length; i++)</pre>
                                                                                                                    Utf8JsonWriter utf8JsonWriter = t writer ??= new Utf8JsonWriter(pipeWriter, new JsonWriterOption
              var randomNumber = _random.Next(1, 10001);
                                                                                                                    utf8JsonWriter.Reset(pipeWriter);
              updateCmd.Parameters.Add(new NpgsqlParameter<int>(parameterName: ids[i], value: results[i].Id));
                                                                                                                    // Body
              updateCmd.Parameters.Add(new NpgsqlParameter<int>(parameterName: randoms[i], value: randomNumber));
                                                                                                                    JsonSerializer.Serialize<World[]>(utf8JsonWriter, rows, SerializerOptions);
              results[i].RandomNumber = randomNumber;
                                                                                                                    // Content-Length
                                                                                                                    lengthWriter.WriteNumeric((uint)utf8JsonWriter.BytesCommitted);
           await updateCmd.ExecuteNonQueryAsync();
```

return results;

Fortunes

```
public async Task<List<Fortune>> LoadFortunesRows()
    var result = new List<Fortune>(20);
    using (var db = new NpgsqlConnection( connectionString))
       await db.OpenAsync();
       using (var cmd = new NpgsqlCommand("SELECT id, message FROM fortune", db))
       using (var rdr = await cmd.ExecuteReaderAsync())
            while (await rdr.ReadAsync())
                result.Add(new Fortune
                    id:rdr.GetInt32(0),
                    message: rdr.GetString(1)
               ));
    result.Add(new Fortune(id: 0, message: "Additional fortune added at request time." ));
   result.Sort();
    return result:
```

private void OutputFortunes(PipeWriter pipeWriter, List<Fortune> model) var writer = GetWriter(pipeWriter, sizeHint: 1600); // in reality it's 1361 writer.Write(fortunesPreamble); var lengthWriter = writer; writer.Write(contentLengthGap); // Date header writer.Write(DateHeader.HeaderBytes); var bodyStart = writer.Buffered; // Body writer.Write(fortunesTableStart); foreach (var item in model) writer.Write(fortunesRowStart); writer.WriteNumeric((uint)item.Id); writer.Write(fortunesColumn); writer.WriteUtf8String(HtmlEncoder.Encode(item.Message)); writer.Write(_fortunesRowEnd); writer.Write(fortunesTableEnd); lengthWriter.WriteNumeric((uint)(writer.Buffered - bodyStart)); writer.Commit();

Benchmarks: Summary

- Logic common to every benchmark:
 - request header parsing
 - request routing
 - response header generation
- Logic common to every DB benchmark:
 - database connection pool
 - random number generation
 - object-relational mapper (ORM)
- Only two benchmarks don't serialize output to JSON
- Utf8 everywhere
- Platform benchmarks are more optimal but also very hacky

Round 18 (July 2019, .NET Core 3.1): JSON

owing 215 of 220 frameworks.				Show filters p
,				
		Test types		Hardwa
posite scores JSON serialization	Single query Multiple q	eries Cached queries	Fortunes Data updates	Plaintext Physical
UN Serializati	ion			
Best (bar chart) Data table	Latency	Framework overhead		
	Best JSON res	ponses per second, Dell R44	0 Xeon Gold + 10 GbE (335 tests)	5 0 1 0 T
k Framework	Best performance (hig	(her is better)	100.00	Errors Cls Lng Plt FE Ao:
1 ulib-json_nt	1,500,569		100.0%	Der Cas Non ULI Lin
2 Durber	1,504,133		97.8%	Mrr Das Bas Han Lin
ibreactor	1,358,605 1		99.4%	Mar C Non Non Lin
s 🖬 🏶 actix	1,357,798		99.4%) Mor Rus Non act Lin
toklo-minihttp	1,357,168		99.3%	Mcr Rus Rus tok Lin
🔳 🏶 actix-raw	1,356,789		99.3%	Pit Rus Non act Lin
thruster	1,355,558		99.2%	Mcr Rus Rus Non Lin
httpbeast	1,353,732		99.1%	Pit Nim Non Non Lin
ester	1,352,096 1		98.9%	Mar Nim Non Non Lin
wizzardo-http	1,351,780		98.9%	Mcr Jav Non Non Lin
= h20	1,550,457		98.8%	PE C Non h2o La
	1,550,052		98.8%	IN Cas Non Non Lin
	1,246,191		90.779	Mir Izy lity Non Lit
firenio-htto-lite	1.341.612		98.2%	Ptt Jav fir Nos Lie
may-minihttp	1,339,396		98.0%	Mar Rus Rus may Lin
= light-4i	1,335,693		97.7%	O Ptt Jav lig Non Lie
undertow	1,333,977		97.6%) Pit Jav Utw Non Lin
a rapidoid-http-fast	1,332,349		97.5%) Pit Jav Rap Non Lie
firenio	1,331,882		97.5%) Ptt Jav fir Non Lin
wertx	1,328,626		97.2%	O Pit Jav ver Non Lie
s 🗰 ceitit	1,327,486		97.1%	Mar Clj Utw Non Lin
i netty	1,322,192		96.8%	Ptt Jav Nty Non Lin
officetioor-raw	1,520,710		96.0%	PE Jav off woo La
rapidoid	1,51/,944		95.4%	Fit Jav Map Non Lit
atraunn-easilisnn-nrefnrir	1,313,130		90.1%	Pit Go Non Non Lin
atreugo-orefork	1,298,360		95.0%	PE Go Non Nos Lis
atreugo-sison-prefork	1,292,709		94.6%) Pit Go Non Non Lin
A spcore-rhtx	1,286,433		94.1%	O Pit C# .NE kes Lie
colossus	1,273,962		93.2%) Mor Sca Akk Non Lin
swoole	1,255,868		91.9%	PE PHP swo Non Lin
blaze	1,234,443		90.3%) Mor Sca bla Non Lin
apronto	1,222,111		89.4%	O Mor Py Non Non Lie
fasthttp-prefork	1,199,319		87.8%	Ptt Go Non Non Lin
openresty	1,199,187		87.8%	Ptt Liss OpR mpx Lie
ateph	1,158,517		84.8%	Bud Cu New New Li
ardy	1,133,334		94.0%	Mar IS war Non Li
servicet	1,144,820 1		83.8%	Pit Jay Syt Res Lin
= voovan	1,129,472		82.7%	PE Jav voo Non Lin
a facillo	1,082,140		79.2%	Mcr C Non Non Lin
vertx-web	1,070,220		78.3%	Mcr Jav ver Nos Lie
cutelyst	1,031,762		75.5%	O Full C++ cut Non Lin
greenlightning	1,027,713		75.2%	Micr Jav Non Non Lie
warp	1,025,313		75.0%	Mor Hid. Wol wai Lin
warp-mysql-haskell	1,024,083		74.9%	Mor Hit Wai wai Li
warp-hasql	1,023,511		74.9%	Mcr Hid. Wai wai Li
blade	1,015,846		74.3%	Fut Jav Non Non Lie
Tasthttp	1,011,449		74.0%	Ptz Go Non Non Lin
atreugo-easyjson	997,931		73.0%	Pit Go Non Non Lin
	991,529		72.5%	
eureugo-sison	AA0'252 IIII		12.376	Pit to men Non Lin
I = (1) 2000000	060 717 1		71.0%	Det Cat ME have the







969,717

+ 36% boost needed

https://www.techempower.com/benchmarks/#section=data-r18&hw=ph&test=json

Round 18 (July 2019, .NET Core 3.1): Fortunes

	Best fortune	es responses per second, Dell R440 Xeon Gold + 10 GbE (339 to	ests)		
Rnk Framework	Best performance	(higher is better)	Errors	Cls	Lng
1 🔳 🗇 <u>actix-core</u>	702,165	100.0%	0	Plt	Rus
2 🔳 🗇 <u>actix-pg</u>	632,672	90.1%		Mcr	Rus
3 ■ <u>h2o</u>	456,058	65.0%	0	Plt	С
4 🗖 atreugo-prefork-quicktemplate	435,874	62.1%		Plt	Go
5 vertx-postgres	403,232	57.4%	0	Plt	Jav
6 📕 <u>ulib-postgres</u>	359,874	51.3%		Plt	C++
7 g reenlightning	341,347	48.6%	0	Mcr	Jav
8 cpoll_cppsp-raw	326,149	46.4%		Plt	C++
9 🗖 atreugo-quicktemplate	319,256	45.5%	0	Plt	Go
10 🗖 go-pgx-prefork-quicktemplate	308,337	43.9%		Plt	Go
11 swoole	307,673	43.8%	0	Plt	PHP
12 🔳 🗇 aspcore-ado-pg	300,613	42.8%	1	Plt	C#

+ 2.5% boost needed

https://www.techempower.com/benchmarks/#section=data-r18&hw=ph&test=json

Round 19 (February 2020, .NET Core 3.1): JSON

2020-05-28 A totally un	velated nineteen		
Round 19 Fast foundation environments at	al software can cure the disease of slow applicatio ind holistic scoring of framework performance acros	ns. New in Round 19 is composite scoring, providing is all test types.	
howing 215 of 220 frameworks.			Show filters panel
nposite scores JSON serialization	Single query Multiple queries Cached que	ries Fortunes Data updates	Plaintext Physical Close
ION serializati	ion		
Best (bar chart) Data table	Latancy Framework overhead		
	Best JSON responses per second, D	ell R440 Xeon Gold = 10 GbE (411 tests)	
zk Framework	Best performance (higher is better)	100.04	Errors Cis Lng Pit FE Aos I
2 ulip-ison ft	1,653,039	100.0%	Pit C++ Non ULI Lin Re
3 pice.v	1,617,611	97.8%	Mar v Non Non Lin Re
4 Libreactor	1,609,461	97.4%	Mor C Non Lin Re
5 may-minitetp	1,593,722	96.4%	Mar Ras Rus may Lin Re
7 wizzardo-http	1 584 775	95.95	Mar Jav Non Non Lin Re
8 tunt	1,575,561	95.3%	Pit D Non Non Lin Re
9 actix-raw	1,573,980	95.2%	Piz Ras Non act Lin Re
0 Uthium	1,567,654 1	94.8%	May (Non Non Lin Re
International postgres	1,567,044	94.0%	Mar C Los bes Lin Ba
13 m h2o	1,551,861	93.9%	O Pitz C Non h3s Lin Re
14 hlacr	1,549,681	93.7%	Mar Cry Non h2o Lin Re
LS Efirenio	1,495,747	90.5%	Pit Jav Sr Non Lin Ba
16 yooyan	1,469,323	88.9%	Pill Jay was Non Lin Re
I anonahorn	1437629	870%	Pit Ket pre Non Lin Re
9 scalene	1,405,554	85.0%	Mar Sca Non Lin Re
10 📕 reitit	1,398,554	84.6%	Mar Clj Ubw Mon Lin Re
1 m httpbeast	1,395,234	84.4%	O PE Nim Non Lin Re
12 Testrictp-easy(son-prefork 11 m the settle	1,556,555	80.8%	Mr Bas Nos at Lin Ba
24 and rapidold	1.327.206	80.3%	Pit Jav Rap Non Lin Re
5 fasthttp-sjson-prefork	1,309,516	79.2%	Pit Go Non Lin Be
16 arapidoid-http-fast	1,307,483	79.1%	Pit Jav Rap Non Lin Be
27 thruster	1,307,463	79.1%	Mer Ras Ras Non Lin Re
 active approximation 	1,293,855	78.3%	Ful Jay Uby Non Lin Re
0 proteus	1,284,728	77.7%	Mar Jev Utw Non Lin Re
1 atreugo-sjson-prefork	1,281,132	77.5%	Piz Go Non Lin Re
12 php:ngx	1,276,244	77.2%	Piz PHP npx npx Lin Re
i joory 14 m truner	1,271,803	75.6%	Hr Ras Ras Hog Lin Be
IS Uphe-H	1,247,431	75.5%	Piz Jav Lig Non Lin Ba
16 jooby-jaxrs	1,247,174	75.4%	Full Jer Utw Non Lin Re
7 • • aspcore-rhtx	1,242,934	75.2%	Pit CF ME kes Un Re
10 KDODY 19 simos	1,239,070	74.9%	My PHP sws Non Lin Re
0 atreugo-prefork	1,224,589	74.1%	Piz Go Non Non Lin Be
1 afasthetp-prefork	1,203,311	72.8%	PE Go Non Non Lin Re
2 officefuor-raw	1,202,663	72.7%	Piz Jav aff woo Lin Re
14 iester	1,192,446	72.1%	Nor Nin Non Non Lin Re
5 coencesty	1,180,321	71.4%	O Pit Lue Opit rige Lin Re
6 undertow	1,170,760	70.8%	Pit Jer Ubr Non Lin Re
7 fiber-prefork	1,133,264 1	68.5%	Piz Go Non Lin Re
	1,119,519	67.7%	Ful Jav ver Non Lin Re
0 = 0 drogon	1,077,826	65.2%	Ful C++ Non Non Lin Re
		65.0%	Mar Lua Lea Lea Lin Re
11 Essan-lua	1,074,255		
11 bitan-lua 12 bitan-lua	1,074,255	64.7%	Pit PHP war Non Lin Re
1 Lisan-Lua 2 workerman 3 Colossus 4 Plane	1,074,255 1,070,023 1,042,956 1,042,956	64.7% 63.1% 62.4%	Pit PHD way Non Lin Ba Mar Sca Akk Non Lin Ba
1 Examples 2 Examples 3 Ecclosses 4 Eblaze 5 Ecclosses 5 Ecclosses 5 Ecclosses 5 Ecclosses 6 Ecclosses 6 Ecclosses 6 Ecclosses 6 Ecclosses 7 Ecclosse	1,074,255 1,070,023 1,042,956 1,031,646 1,021,544	64.7% 63.1% 62.4% 62.2%	Pitt PHD warr Non Lin Re Nitr Sca Akit Non Lin Re Nitr Sca Akit Non Lin Re Nitr Sca Nita Non Lin Re Nitr C++ Nion Non Lin Re
1 Exam-Lus 2 exorkerman 3 colossus 4 eblase 5 extpp-async 6 extpl-minihttp	1,074,235 1,070,025 1,042,956 1,051,646 1,028,148 1,028,547	64.7% 63.1% 62.4% 62.2% 63.2%	PBI PHD warr Non Lin Rum Natr Sca Abk Non Lin Rum Natr Sca Abk Non Lin Rum Natr Sca Abk Non Lin Rum Natr Sca Bak Non Lin Rum Natr Firet Rum Non Lin Rum
1 Examples 2 Workerman 3 Collossis 4 Distat 5 catgo-async 6 External 7 warp-hasgi	1,074,255 1,070,033 1,042,956 1,031,646 1,022,547 994,992	64.7% 63.1% 62.4% 62.2% 61.2% 60.3%	PE PHF wave Name Line Name Marr Sca Able Name Line Name Marr Sca Able Name Line Name Marr Sca Able Name Line Name Marr Sca Rame Marr Line Name Marr Hall Ward wave Line Name
1 Isam:Sua 2 workerman 3 0.000596 4 blace 5 astap-async 6 blace-ninimp 7 warp-hasgi 4 sam_mysgi-hasketi 9 warp-hasgi	1074255 1070025 104235 103144 103144 1022547 99.592 99.763 99.763	64.7% 63.1% 62.4% 62.2% 60.2% 60.2% 60.2%	Ph PHP and Max Lin Max No Sca Max Max Lin Max No Sca Max Max Lin Max No Sca Max Max Lin Max No C+ Na Max Lin Max No C+ Na Max Lin Max No He Na Max Lin Max No He Max Max Lin Max No He Max Max Lin Max
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii	1074255 1074255 104256 1031144 1001244 1001247 995.90 905.90 9	64.7% 63.1% 62.4% 61.2% 60.3% 60.2% 60.2% 60.2% 60.2%	PD PDF unset Num Link Num Num Sca Sca Num Sca Num Num Num Sca Sca Num Num Num Num Num Num Sca Sca Num
	1,074,255 1,042,556 1,031,444 1,021,544 1,022,547 1,025,547	64.7% 63.1% 62.2% 62.2% 60.3% 60.2% 60.2% 60.2% 60.2% 60.0% 60.0% 60.0%	HI HI<
	2074255 207021 205244 2020144 2020247 996592 997592 996592 997592 99759 9	4.7% 4.1% 4.2%	NP NP<
Eleministra Eleministra Ecologia Ec	1.07435 1.04435 1.04435 1.00144 1.00244 1.00244 1.00244 1.00244 1.00244 1.00244 1.0099 1.009	(4.7%) (5.1%) (5.2%) (4.2%) (4.2%) (4.5%)	Ph PH ear So
# Bean-Luba # # conterman # state	10/435 10/003 10/156 10/164 10	(4.7%) 4.3.1% 4.2.4% 4.2.5% 4.2.5% 4.0.5%	N NO and bit No So No So No So No So No No <th< td=""></th<>
	1.0/425 1.0/425 1.0/025 1.0/025 1.0/024 1.0/02	4478 6318 6226 6128 6128 6128 6238 6238 6238 6228 62588 6258 6258 6258 6258 6258 6258 6258 6258 6258 6258	N NO NO </td
	10/335 1 10/335 1 10/346 1 10/346 1 10/346 1 10/347	4478 4478 424 424 424 424 424 425 422 42 42 42 42 42 4 4	Pit Pit
	1.03.151 1.03.154 1.03.1	6478 6158 6246 6256 6276 6078	N N



+ 32.3% boost needed

https://www.techempower.com/benchmarks/#section=data-r19&hw=ph&test=json

Round 19 (February 2020, .NET Core 3.1): Fortunes

Fortunes

	Best fortunes res	sponses per second, Dell R440 Xeon Gold + 10 GbE (424 tes	sts)									
Rnk Framework	Best performance (high	her is better)	Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA
1 🗖 🗇 drogon-core	678,278	100.0%		Ful	C++	Non	Non	Lin	Pg	Lin	Raw	Rea
2 actix-core	651,144	96.0%		Plt	Rus	Non	act	Lin	Pg	Lin	Raw	Rea
3 🔳 🗇 <u>actix-pg</u>	607,052	89.5%		Mcr	Rus	Non	act	Lin	Pg	Lin	Raw	Rea
4 🔳 🗇 <u>drogon</u>	553,366	81.6%		Ful	C++	Non	Non	Lin	Pg	Lin	Mcr	Rea
5 📕 may-minihttp	476,965	70.3%		Mcr	Rus	Rus	may	Lin	Pg	Lin	Raw	Rea
6 ■ <u>h2o</u>	411,176	60.6%		Plt	c	Non	h2o	Lin	Pg	Lin	Raw	Rea
7 Iithium-postgres	401,783	59.2%		Mcr	C++	Non	Non	Lin	Pg	Lin	Ful	Rea
8 📕 fasthttp-prefork-quicktemplate	363,587	53.6%		Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea
9 🔳 atreugo-prefork-quicktemplate	362,342	53.4%		Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea
10 ■ <u>hyper-db</u>	358,511	52.9%		Mcr	Rus	Rus	Нур	Lin	Pg	Lin	Raw	Rea
11 ■ php-ngx-pgsql	356,507	52.6%		Plt	PHP	ngx	ngx	Lin	Pg	Lin	Raw	Rea
12 🗖 workerman-pgsql	352,508	52.0%		Plt	PHP	wor	Non	Lin	Pg	Lin	Raw	Rea
13 vertx-postgres	347,356	51.2%		Plt	Jav	ver	Non	Lin	Pg	Lin	Raw	Rea
14 📕 <u>ulib-postgres</u>	344,634	50.8%		Plt	C++	Non	ULi	Lin	Pg	Lin	Mcr	Rea
15 🗖 fasthttp-quicktemplate	319,764	47.1%		Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea
16 🗖 atreugo-quicktemplate	319,390	47.1%		Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea
17 g reenlightning	318,601	47.0%		Mcr	Jav	Non	Non	Lin	Pg	Lin	Raw	Rea
18 🗖 jooby-pgclient	312,439	46.1%		Ful	Jav	Utw	Non	Lin	Pg	Lin	Raw	Rea
19 Efiber-prefork	301,604	44.5%		Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea
20 So-pgx-prefork-quicktemplate	298,935	44.1%		Plt	Go	Non	Non	Lin	Pg	Lin	Raw	Rea
21 ■ <u>lithium</u>	296,750	43.8%		Mcr	C++	Non	Non	Lin	Му	Lin	Ful	Rea
22 workerman	291,339	43.0%		Plt	PHP	wor	Non	Lin	Му	Lin	Raw	Rea
23 php-ngx-mysql	290,312	42.8%		Plt	PHP	ngx	ngx	Lin	Му	Lin	Raw	Rea
24 🔳 💎 aspcore-rhtx-pg	285,398	42.1%		Plt	C#	.NE	kes	Lin	Pg	Lin	Raw	Rea
25 swoole	283,728	41.8%		Plt	PHP	swo	Non	Lin	Му	Lin	Raw	Rea
26 🔳 🗇 aspcore-ado-pg	273,121	40.3%		Plt	C#	.NE	kes	Lin	Pg	Lin	Raw	Rea

+ 31.2% boost needed

https://www.techempower.com/benchmarks/#section=data-r19&hw=ph&test=fortune

Measure, Measure, Measure

How to run the benchmarks?

- Sébastien Ros Modern BCL talk: <u>https://msit.microsoftstream.com/video/95878f4d-6e5b-4655-850e-5056fe92f119</u>
- DOC: <u>https://github.com/aspnet/Benchmarks/blob/master/scenarios/README.md</u>

dotnet tool install Microsoft.Crank.Controller --version "0.1.0-*" –global crank --config https://raw.githubusercontent.com/aspnet/Benchmarks/master/scenarios/platform.benchmarks.yml --scenario plaintext --profile aspnet-citrine-lin

To profile and get a trace file: --application.collect true

To use given .dll in the publish app: --application.options.outputFiles \$pathToFile.dll

Alternative: run them locally

- git clone https://github.com/aspnet/Benchmarks.git
- Start the web server:
 - cd Benchmarks/src/BenchmarksApps/Kestrel/PlatformBenchmarks
 - dotnet run -c Release
 - You can publish a self-contained version and replace *.dll files if you want to test your local changes
- Start the HTTP benchmarking tool:
 - cd Benchmarks/src/WrkClient
 - chmod +x wrk
 - ./wrk -c 1 http://127.0.0.1:8080/plaintext -d 1m -t 1 --header
 "Accept:text/plain,text/html;q=0.9,application/xhtml+xml;q=0.9,application/xml;q=0.8,*/*;q=0.7" -s scripts/pipeline.lua 16
 - (it's a sample command, don't forget to change the number of connections, duration and thread count)
 - The magic header value comes from wrk.yml

The beginning of a performance investigation

Update Back Forward Totals Metric: 178,610.0 Count: 178,546.0 First: 16,334.435 Last: 31,547.770 Last-First	t: 15,213.33	35 Metri	c/Interval:	11.74 Tim	eBucket: 4
<u>Start:</u> 15,939.025 Y End: 31,878.051 Y Find: Json					
GroupPats: [no grouping] Y Fold%: Y FoldPats:			×	IncPats:	Process%
By Name ? Caller-Callee ? CallTree ? Callers ? Callees ? Flame Graph ? Notes ?					
Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % <u>?</u>	Inc ?
kernel.kallsyms!_raw_spin_unlock_irgrestore	5.6	10,034	10,036	5.6	10,056.3
kernel.kallsyms!finish_task_switch	4.4	7,839	7,835	5.4	9,664.6
kernel.kallsyms!softirgentry_text_start	2.0	3,657	3,654	21.7	38,780.3
libpthread-2.28.so!_libc_recvmsg	2.0	3,483	3,480	10.9	19,553.9
libcoreclr.so!Object::TryEnterObjMonitorSpinHelper	1.7	2,970	2,967	2.2	3,945.9
kernel.kallsyms!do_syscall_64	1.4	2,516	2,513	29.0	51,712.0
libcoreclr.so!JIT_MonExit_Portable	1.3	2,349	2,347	1.7	3,096.0
Microsoft. A sp Net Core. Server. Kestrel. Core! Microsoft. A sp Net Core. Server. Kestrel. Core. Internal. Http://www.core.com/abs/abs/abs/abs/abs/abs/abs/abs/abs/abs	1.1	2,038	2,036	37.2	66,495.5
System. Private. CoreLib! System. Runtime. Compiler Services. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Compiler Services. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Compiler Services. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Compiler Services. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Compiler Services. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Compiler Services. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Compiler Services. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Class (void*, object) and the system. Cast Helpers:: Is Instance Of Cl	1.1	1,950	1,948	1.5	2,603.6
libpthread-2.28.so!_libc_sendmsg	1.0	1,860	1,858	17.5	31,202.6
System.Private.CoreLib!System.Collections.Concurrent.ConcurrentQueueSegment`1[SystemCanon]::TryD	0.9	1,671	1,669	1.5	2,676.6
ld-2.28.so!_tls_get_addr	0.9	1,611	1,609	1.2	2,072.1
libcoreclr.so!JIT_WriteBarrier	0.9	1,537	1,535	1.2	2,108.1
System.Private.CoreLib!System.Collections.Concurrent.ConcurrentQueueSegment`1[SystemCanon]::TryE	0.8	1,448	1,446	1.2	2,123.1
libc-2.28.so!epoll_wait	0.8	1,364	1,363	5.6	9,940.6
kernel.kallsyms!tcp_recvmsg	0.7	1,333	1,331	3.9	6,882.5
System. Private. Core Lib! System. Runtime. Compiler Services. Cast Helpers:: Is Instance Of Interface (void*, object) and the system. System. Private is the system of the system of the system of the system of the system. The system of the system. The system of the system. The system of the system. The system of the syst	0.7	1,322	1,321	1.0	1,817.8
kernel.kallsyms!_fget	0.7	1,320	1,318	1.0	1,699.9
kernel.kallsyms!_nf_conntrack_find_get	0.7	1,290	1,288	0.8	1,454.7
Microsoft. A spNetCore. Server. Kestrel. Core! Microsoft. A spNetCore. Server. Kestrel. Core. Internal. Http: HttpParserver. Kestrel. Core. Server. Server. Kestrel. Core. Server. Kestrel. Core. Server. Kestrel. Core. Server. S	0.7	1,236	1,235	2.5	4,437.4
kernel.kallsyms!ixgbe_clean_rx_irq	0.7	1,214	1,213	17.8	31,825.1
System.Text.Json!System.Text.Json.JsonSerializer+ <writeasynccore>d_65::MoveNext()</writeasynccore>	0.7	1,180	1,179	14.6	26,047.7

B2	0	•	X 🗸	fx.	=SU	M(A1:A20)
	А	В	С	[D	E
1	5.6					
2	4.4					
3	2					
4	2					
5	1.7					
6	1.4					
7	1.3					
8	1.1					
9	1.1					
10	1					
11	0.9					
12	0.9					
13	0.9					
14	0.8					
15	0.8					
16	0.7					
17	0.7					
18	0.7					
19	0.7					
20	0.7	29.4				
			T			

Maybe Flame Graph can tell us something?



What if we fold All Threads?



Can we use Concurrency Visualizer?

Sha	reSour	ce_2018-11-16_135335.CvTra	ace ⊹> >	ShareSource_2018-11-16_132	123.CvTrace	App.g.i.cs	App.xaml	.cs				-
Υl	ير المنتخر		1. Carlor of the second	المرابع كمايك فيناب ومساود معير ومنا	Annual Contraction of	And a second second		I have been been been been been been been be	N. Parter	A Manager	and the second se	
	Utiliza	ation Threads Co	ores									Demystify
żz	oom		Sort by:	Start Time - Markers	• + + T	₩ ₩2 ₩2 ₩						-
			Seconds	1								
Th	read ID) Name	0	20	40	60	80		100	120	140	160
22	500	Main Thread										
16	080	Thread Pool										
10	748	Thread Pool										
12	556	Thread Pool										
63	76	Debugger Helper										
70	68	CLR Worker Thread										
16	704	Worker Thread										
15	668	Thread Pool										
21	480	Thread Pool										
20	772	Worker Thread							_			
		Worker Thread										
			<u>г</u>									
	0.9/	Visible Timeline Profile		Profile Report	t 🗐 Unblocking	g Stack 🕑 H	lints					-
	76%	Synchronization	E	xecution Noise redu	ction at 2_%	✓ Just	My Code					Export
	11%			1	lame		Inclusive Samp	Exclusive Sam	Inclusive (Perc	Exclusive (Per	: Details	
Ы	9%	Sleep		▶ ntdll.dll			37	0	32.46%	0.00%		
	4%	Memory Managem	nent	Cirjit.dll			19	0	16.67%	0.00%		
	0%	Preemption		ntoskrni.exe			1/	0	12.20%	0.00%		
	0%	UI Processing		 A sharesource eve 			7	0	6 1/1%	0.00%		
		Per Thread Summary		✓ sharesource.exe!SDKTem	plate.Program.Mai	'n	7	0	6.14%	0.00%	ShareSource!SDKTemplate.Pr	ogram.Main:Apr
		Disk Operations		windows.ui.xaml.ni.dll	0x55ff52		4	0	3.51%	0.00%	Windows.UI.Xaml.ni!0x55ff52	
		Markers		b windows.ui.xaml.ni.dll	0x55ffc2		3	0	2.63%	0.00%	Windows.UI.Xaml.ni!0x55ffc2	
	0%	This Process		combase.dll			3	0	2.63%	0.00%		
DG	1%	Other Processes		unknown			3	0	2.63%	0.00%		
	0%	Paging		b user32.dll			3	0	2.63%	0.00%		
				wdfilter.sys			3	0	2.63%	0.00%		
				4								
												P

Is there any other tool that we could use?

Converter for Trace Event json (Chrome as viewer) #447

Closed benaadams opened this issue on 25 Aug 2019 · 11 comments

_	
0.000	The second
100	1.2
	0

benaadams commented on 25 Aug 2019 • edited 👻

Member 😳 …

@rickbrew mentioned on twitter he'd added tracing to Paint.NET using the tracing format used by Chrome (and Brave and the new Microsoft Edge) dev tools.

It would be good to add this as and output/conversion format for dotnet-trace as either browsers UI are highly developed and well used and understood especially by webdevs and it can happily open and analyse file in that format from other sources.

Trace Event spec

chrome://tracing can open gzipped files (.json.gz) where as dev tools only understands them as raw .json . Dev tools is a little easier to use (and probably most familiar) whereas tracing is a dedicated UI for looking at larger traces. So it may be useful to provide an additional .gz output option, but not have it as the default.

The json is also a format speedscope understands

/cc @migueldeicaza @lucasmeijer





Needs a little polish, but it went together pretty quickly. Won't open in edge://tracing , but opens in devTools (haven't tried Chrome yet). Very cool timeline view of stacked icicle graphs per thread!

https://github.com/dotnet/diagnostics/issues/447 https://github.com/microsoft/perfview/pull/1113

How to use it?

PerfView



Chromium



Much better Overview!

0.5 ms	1.0 ms	1.5 m	15 2.0 m	ns 2.5 r	ms 3.0 n	ns 3.5	ms 4.0 ms	s 4.5 m	is 5.0 n	ns 5.5 r	ms 6.0 m	s 6.5 i	ms 7.1	0 ms 7.	5 ms 8.0) ms 8.5	ms 9.0	ms 9.5 r	ms 10.0	ms 10.5	ims 11.0	ms 11.5 ms	12.0 m	is 12.5 n	ns 13.0) ms 13.5	ms 14.0	ms 14.5	ms 15.0 ms	
	. <i>tellin</i> s		n in service s	WW		E SALANGE MANDER	alpantika a di					a ann a mhli			- 10	Monde and and a start	all a sin	dil e un ann			a di a di a di	an ann a	·				nnin, Y			
0.5 ms	1.0 ms	1.5 ms	s 2.0 ms	s 2.5 m	s 3.0 ms	s 3.5 m	4.0 ms	4.5 ms	5.0 ms	5.5 ms	6.0 ms	6.5 ms	7.0 m	ns 7.5 m	ns 8.0 m	s 8.5 ms	9.0 ms	9.5 ms	10.0 ms	10.5 ms	11.0 ms	11.5 ms	12.0 ms	12.5 ms	13.0 ms	13.5 ms	14.0 ms	14.5 ms	15.0 ms	
Thread 31846																														4
Thread 31667																														۰.
► Thread 31900																														J.
Thread 31840																														Ł.
 Thread 31842 															1.1.1.1															
► Thread 31902							∬ □ Ϊ [] []																							1
 Thread 31849 Thread 21826 										╙╌┛╝┪┪┩				▞▖¿ᡗ᠊᠊᠓᠆ᡁᢂ	-												.,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,			4
 Thread 31905 													1 							111										Λ.
► Thread 31899																														1
 Thread 31895 Thread 21671 																														4.
 Thread 31903 																														í"
► Thread 31901								التبلقا البل																						i.
Thread 31843																														
Thread 31850																														٥.
► Thread 31854																														
 Thread 31857 Thread 31861 																														r.
▶ Thread 31862						d në dë në m ë të							i ii in iii	0.0011.00						11111111	II 'IIII 'I						117 1 1 (11)			i.
Thread 31598																														
 Thread 31599 Thread 31600 																														
▶ Thread 31601																														
 Thread 31602 Thread 31602 																														
 Thread 31604 																														
▶ Thread 31605																														
 Thread 31606 Thread 31607 																														
 Thread 31608 																														
► Thread 31609																														
 Thread 31341 Thread 31444 																														
▶ Thread 31560																														
Thread 16																														
 Thread 31544 Thread 22 																														
▶ Thread 31946																														
 Thread 31554 Thread 21559 																														
 Thread 31545 																														
▶ Thread 31327																														
Thread 31328 Thread 31328																														
► Thread 31331																														
Thread 31332																														
 Thread 31333 Thread 31334 																														
▶ Thread 31335																														
 Thread 31336 Thread 31230 																														
 Thread 31338 Thread 31330 																														
▶ Thread 31337																														
 Thread 31339 Thread 31233 																														
 Thread 31323 Thread 31952 																														
▶ Thread 31947																														

Is GC a problem? No.



Why do we have few threads that are not 100% active?

0.5 ms	1.0 ms	1.5	ms 2/	0 ms 2.5	ms 3.0 n	ns 3.5 n	ns 4.0 ms	4.5 m	s 5.0 m	s 5.5 ms	6.0 ms	0.5 ms	7.0 ms	7.5 ms	8.0 ms	8.5 ms 9.0 n	ms 9.5 m	ns 10.0 m	ns 10.5 ms	s 11.0 ms	11.5 m	is 12.0 r	15 12.5	5 ms 13.	0 ms 13.	5 ms 14.0	ms 14.5 n	ms 15.0 ms
1977 - S.	. hulling it		<i>₩</i> ₩. * * *	· · · · · · · · · · · · · · · · · · ·			e de ante de contra d					···· • ···		<i></i>	and the state of the second	an alfaller an ann a	ad a sur ann			an a	ar sparse			88 		inder V	10	
0.5 ms	1.0 ms	1.5 m	ns 2.0	ms 2.5 m	is 3.0 ms	s 3.5 ms	s 4.0 ms	4.5 ms	5.0 ms	5.5 ms	6.0 ms	6.5 ms	7.0 ms	7.5 ms	8.0 ms 8.1	5 ms 9.0 ms	9.5 ms	10.0 ms	10.5 ms	11.0 ms	11.5 ms	12.0 ms	12.5 ms	13.0 ms	13.5 ms	14.0 ms	14.5 ms	15.0 ms
Thread 31846							/																					
Thread 31841																												
Inread 31667 Thread 31900									1111																			
Thread 31840	the th e									1111							1 I I I I I I I I I I I I I I I I I I I										1111111	
► Thread 31672																												
Thread 31842										1.0000000000000000000000000000000000000																		
Thread 31902																												
Thread 31849										╵╴╹╵┵╍┚╴╴╎╴																		
Inread 31030 Thread 31905													╎╻╻╏╷┛╹╎╎╎╷╵╹╴╴┧															
Thread 31899										111111	<u> 11100 11 11 11 11 1</u>																	
► Thread 31895																												
Thread 31671																												
Thread 31903																												
Thread 31901																												
 Thread 31835 					▖▋▋▖▖▝▝▝▝▝▖▖▖▖																							
Thread 31850																												
► Thread 31854						0.000000	ADDONÉCÍCÍ ČÍ) [[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[n i i inini i nin					lii mi iii	1			
Thread 31857						, , , , , , , , , , , , , , , , , , ,							nol		Mait					UUUU (0)								
Thread 31861													-pu		varu													
Thread 31862	1111	1 11 11 11					10 1111						111-111	1 + + + + + + + + + + + + + + + + + + +													1 1 10111	
Thread 31599																												
► Thread 31600																												
Thread 31601																												
Thread 31602																												
Thread 31603																												
Thread 31604																												
▶ Thread 31606																												
▶ Thread 31607																												
Thread 31608																												
Thread 31609																												
Thread 31341 Thread 31444																												
Thread 31560																												
► Thread 16																												
▶ Thread 31944																												
Thread 22																												
 Inread 31946 Thread 31554 													in the second															
Thread 31558																												
▶ Thread 31545																												
▶ Thread 31327																												
Thread 31328																												
 Inread 31329 Thread 31331 																												
Thread 31332																												
► Thread 31333																												
► Thread 31334																												
Thread 31335																												
 Inread 31336 Thread 31338 																												
 Thread 31330 																												
Thread 31337																												
► Thread 31339																												
► Thread 31323																												
Thread 31952																												
Thread 31947																												

What is epoll?

what's epoll?

Okay, we're ready to talk about epoll!! This is very exciting to because I've seen epoll_wait a lot when stracing programs and I often feel kind of fuzzy about what it means exactly.

The epoll group of system calls (epoll_create, epoll_ctl, epoll_wait) give the Linux kernel a list of file descriptors to track and ask for updates about whether

Here are the steps to using epoll:

- 1. Call epoll_create to tell the kernel you're gong to be epolling! It gives you an id back
- 2. Call epoll_ctl to tell the kernel file descriptors you're interested in updates about. Interestingly, you can give it lots of different kinds of file descriptors (pipes, FIFOs, sockets, POSIX message queues, inotify instances, devices, & more), but **not regular files**. I think this makes sense pipes & sockets have a pretty simple API (one process writes to the pipe, and another process reads!), so it makes sense to say "this pipe has new data for reading". But files are weird! You can write to the middle of a file! So it doesn't really make sense to say "there's new data available for reading in this file".
- 3. Call epoll_wait to wait for updates about the list of files you're interested in.

Side note: People don't like epoll

more select & epoll reading

I liked these 3 posts by Marek:

- -

- $\circ~$ select is fundamentally broken
- $\circ\,$ epoll is fundamentally broken part 1
- epoll is fundamentally broken part 2

In particular these talk about how epoll's support for multithreaded programs has not historically been good, though there were some improvements in Linux 4.5.



"The Linux Programming Interface" book

63.4.5 Performance of epoll Versus I/O Multiplexing

Table 63-9 shows the results (on Linux 2.6.25) when we monitor N contiguous file descriptors in the range 0 to N - 1 using *poll()*, *select()*, and *epoll*. (The test was arranged such that during each monitoring operation, exactly one randomly selected file descriptor is ready.) From this table, we see that as the number of file descriptors to be monitored grows large, *poll()* and *select()* perform poorly. By contrast, the performance of *epoll* hardly declines as N grows large. (The small decline in performance as N increases is possibly a result of reaching CPU caching limits on the test system.)

For the purposes of this test, FD_SETSIZE was changed to 16,384 in the *glibe* header files to allow the test program to monitor large numbers of file descriptors using *select()*.

Number of descriptors monitored (N)	<i>poll()</i> CPU time (seconds)	<i>select()</i> CPU time (seconds)	<i>epoll</i> CPU time (seconds)
10	0.61	0.73	0.41
100	2.9	3.0	0.42
1000	35	35	0.53
10000	990	930	0.66
4			

Table 63-9: Times taken by poll(), select(), and epoll for 100,000 monitoring operations

In Section 63.2.5, we saw why *select()* and *poll()* perform poorly when monitoring large numbers of file descriptors. We now look at the reasons why *epoll* performs better:

Reducing the epoll threads to 1



#2346: 1 epoll thread per 1024 connections

√ 2		src/libra	ries/System.Net.Sockets/src/System/Net/Sockets/SocketAsyncEngine.Unix.cs 🖺
		@@ -105,7	+105,7 @@ public bool TryRegister(SafeSocketHandle socket, out Interop.Error error)
105	105		//
106	106		<pre>private static readonly IntPtr MaxHandles = IntPtr.Size == 4 ? (IntPtr)int.MaxValue : (IntPtr)long.MaxValue;</pre>
107	107	#endif	
108		-	<pre>private static readonly IntPtr MinHandlesForAdditionalEngine = s_engineCount == 1 ? MaxHandles : (IntPtr)32;</pre>
	108	+	<pre>private static readonly IntPtr MinHandlesForAdditionalEngine = s_engineCount == 1 ? MaxHandles : (IntPtr)EventBufferCount;</pre>

		6/12 cores			14/28 cores		
Benchmark (Median)	Before	After	Diff %	Before	After	Diff %	
Plaintext	1,977,695	2,045,820	3.33%	4,023,529	4,113,391	2.18%	
Json	340,522	382,570	10.99%	763,704	830,055	7.99%	
DbFortunesRaw PostgreSQL	103,392	111,416	7.20%	249,026	274,362	9.23%	
PlaintextNonPipelined	376,117	428,488	12.22%	833,743	851,074	2.04%	

https://github.com/dotnet/runtime/pull/2346

#19396: Add SocketTransportOption to enable/disable WaitForData Benchmark results using Citrine:

Files changed 5

오	Conversation	21

-O- Commits 6 🗊 Checks 17

tmds commented on 27 Feb

This allows to opt-out of the zero-byte read that is performed to reduce memory usage for idle connections. This read has a measurable impact on TE JSON benchmark.

```
// Wait for data before allocating a buffer.
await _receiver.WaitForDataAsync();
if (_waitForData)
{
    // Wait for data before allocating a buffer.
    await _receiver.WaitForDataAsync();
```

// Ensure we have some reasonable amount of buffer space
var buffer = input.GetMemory(MinAllocBufferSize);

https://github.com/dotnet/aspnetcore/pull/19396

JSON

WaitForData enabled:

--jobs "..\Benchmarks\benchmarks.json.json" --scenario "Json"

RequestsPerSecond:	840,803
Max CPU (%):	100
WorkingSet (MB):	417
Avg. Latency (ms):	0.86
Startup (ms):	504
First Request (ms):	44.64
Latency (ms):	0.11
Total Requests:	12,696,202
Duration: (ms)	15,100
Socket Errors:	0
Bad Responses:	0
Build Time (ms):	3,001
Published Size (KB):	26,065
SDK:	5.0.100-preview.2.20120.3
Runtime:	5.0.0-preview.2.20125.16
ASP.NET Core:	5.0.0-preview.2.20126.7

WaitForData disabled:

--jobs "..\Benchmarks\benchmarks.json.json" --scenario "Json"

RequestsPerSecond:

861,744

It was not that simple...



https://github.com/dotnet/runtime/pull/33855 - revert of 1 epoll thread per 1024 connections

Why the Platform benchmark has regressed?

Jpdate Back Forward Totals Metric: 421,025.0 Count: 421,025.0 First: 5,480	.811 Last: 2	20,752.74	44 Last-Fir	st: 15,271.	933 Metri	ic/Interval: 2	7.57 Time	Bucket: 5	01.2	
Start: 5,103.143 Y End: 21,141.593 Y Find: epoll_wait										
GroupPats: [no grouping] Y Fold%:	~]	FoldPats:				Ý	IncPats	.NET Th	readPool .N	IET Server GC dotn Y ExcPats
By Name 2 Caller-Callee 2 CallTree 2 Callers 2 Callees 2 Flame Grap	h ? Note	s <u>?</u>								
Name 2	Exc % 2	Exc 2	Exc Ct 2	Inc % ?	Inc 2	Inc Avg 2	Inc Ct 2	Fold ?	Fold Ct ?	When ?
libcoreclr.so!CLRLifoSemaphore::Wait	19.7	82,976	82,976	29.2	122,830.0	1.0	122,830	0	0	0**************************************
libcoreclr.so!ThreadpoolMgr::WorkerThreadStart	5.6	23,642	23,642	81.0	340,949.0	1.0	340,949	0	C	1**************************************
libcoreclr.so!CLRLifoSemaphore::Release	4.9	20,800	20,800	5.4	22,700.0	1.0	22,700	0	0	OFFECDDFFIIFCBCEFFJFEDCGIECCFHF_
libcoreclr.so!ThreadpoolMgr::MaybeAddWorkingWorker	4.2	17,533	17,533	4.6	19,357.0	1.0	19,357	0	C	OBDCA9AGDGBCAACFEECCAEACDDBBFCD_
System.Private.CoreLib!System.Threading.ThreadPoolWorkQueue::Dequeu	4.0	16,780	16,780	7.1	29,864.0	1.0	29,864	0	0	OJJILLIIJHIJNLMLJHHILIIHLKNKHI_
libc-2.28.so!sched_yield	2.6	11,071	11,071	7.6	31,986.0	1.0	31,986	0	0	oKROMOPERENOLEEREFLNERJJLJLIEMJ_
kernel.kallsyms!_schedule	2.3	9,820	9,820	3.1	12,863.0	1.0	12,863	0	0	178A89A8798978798688880878878880
System.Private.CoreLib!System.Collections.Concurrent.ConcurrentQueueS	2.3	9,708	9,708	3.2	13,525.0	1.0	13,525	0	0	2087A9878768BA9A8779989979AB878_
kernel.kallsyms!do_syscall_64	2.2	9,372	9,372	17.3	73,040.0	1.0	73,040	0	0	B*************************************
System.Private.CoreLib!System.Threading.ThreadPoolWorkQueue::EnsureT	1.6	6,723	6,723	10.5	44,123.0	1.0	44,123	0	C	1TTRQPQVSWVTRRSTTUWRSSPUVUSRWUS_
libcoreclr.so!ManagedPerAppDomainTPCount::TakeActiveRequest	1.6	6,675	6,675	1.7	7,290.0	1.0	7,290	0	0	0454434444655554455454444445454_
System.Private.CoreLib!System.Threading.ThreadPoolWorkQueue::MarkTh	1.3	5,391	5,391	1.4	5,967.0	1.0	5,967	0	0	0343332333334444433343333445333_
libcoreclr.so!ManagedPerAppDomainTPCount::SetAppDomainRequestsAc	1.2	5,190	5,190	7.9	33,219.0	1.0	33,219	0	0	OLMRIJJOMQOLIJJNMNOLJLJNOMJJONM_
libcoreclr.so!JIT_MonExit_Portable	1.2	4,932	4,932	1.3	5,410.0	1.0	5,410	0	0	o4333323333333333333333223344333_
libpthread-2.28.so!_libc_recvmsg	1.1	4,837	4,837	5.4	22,630.0	1.0	22,630	0	0	OEFDDDCDFEDFHIHFDEDEFEDDDGHIEDE_
kernel.kallsyms!softirgentry_text_start	1.1	4,831	4,831	8.2	34,540.0	1.0	34,540	0	0	oMMMCOPELJLNOQQNEERLNNNJLMPPMLM_

← → ♂ ☆	🖸 🔒 https://w	ww.speedscope.app			··· 🗵 🕁	III\ 🖸 🕝	. ≡
д Time Order 🔚 Left H	eavy 🚍 Sandwich		Thread (1397)	7)		T Export 🛐 Import	? Help
2.00s	4.00s	6.00s	8.00s	10.00s	12.00s	14.005	
7							
2.00s	4.00s	6.08s	8,00s	10.00s	12.00s	14.00s	
.NET ThreadPool						enoll w	ait
Thread (13977)							an
libpthread-2.28.so!start_th	read					unknown ! unknown	
libcoreclr.so!CorUnix::CPal	Thread::ThreadEntry					libc-2.28.solepol	l_wait
libcoreclr.so!ThreadNative:	KickOffThread					kernel.kalls.fter	_hwframe
libcoreclr.so!ManagedThread	Base::KickOff					kernel.kalls.o_sys	scall_64
libcoreclr.so!ManagedThread	Base_DispatchOuter					kernel.kalls_s_epc	oll_wait
libcoreclr.so!ThreadNative:	KickOffThread_Worker					kernel.kallsyms!ep	p_poll
libcoreclr.so!MethodDescCal	lSite::CallTargetWorker					kernel.kall_pnstpr	rop.17
libcoreclr.so:CallDescrWork	erInternal					kernel.kall_events	s_proc
System.Private.CoreLib.dll	unknown					kernel.kisra.12	
System.Net.Sockets.dll!unkne	own					kernel_k_poll	
System.Net.Sockets!System.Net	et.Sockets.SocketAsyncContext	::HandleEvents(valuetype Int	erop/Sys/SocketEvents)		Syst_ext)	S_) k_l _	
System.Net.Sockets/System.Net.	et.Sockets.SocketAsyncContext	+OperationQueue`1[SystemO	anon]::HandleEvent(class Sys	tem.Net.Sockets.SocketAsyncC	ontext) l…e …		
System.Net.Sockets!System.Net.Net.Net.Net.Net.Net.Net.Net.Net.Net	et.Sockets.SocketAsyncContext	+AsyncOperation::Dispatch()		libc.	able l_e		
System.Private.CoreLib!Syste	em.Threading.ThreadPool::Uns:	feQueueUserWorkItem(class Sy	stem.Threading.IThreadPoolWo	-kItem,bool)			
System.Private.CoreliblSyste	em.Threading.ThreadPoolWorkO	eue::Enqueue(object_bool)		Sv 1)	nsureThread	Requested	
System.Private.CoreLib!Syste	em.Threading.ThreadPoolWorkQu	eue::EnsureThreadRequested()	575	(10)	insure rin eut	mequesteu	
libcoreclr.so!ThreadPoolNat	ve::RequestWorkerThread						
libcoreclr.so!ThreadpoolMgr	:SetAppDomainRequestsActive						
libcoreclr.so!ManagedPerApp	DomainTPCount::SetAppDomainRe	questsActive					
libcoreclr.so!CLRemaphore:	Release libcoreclr.s.orkin	gWorker					

Kount has provided an excellent explanation

kouvel c	ouvel commented on 20 Mar • edited 👻 Member 😳 …					
It still single	does not answer the question of why replacing a few epoll threads that were enqueueing work on ThreadPool with a one leads to degradation of the performance of enqueueing					
A possibi down the	lity could be that more epoll threads are more frequently able to keep the thread pool busy enough for that path to go fast path:					
runtim Lines 4	e/src/libraries/System.Private.CoreLib/src/System/Threading/ThreadPool.cs 35 to 445 in 5c6a91d					
435	<pre>int count = numOutstandingThreadRequests;</pre>					
436	<pre>while (count < Environment.ProcessorCount)</pre>					
437	{					
438	<pre>int prev = Interlocked.CompareExchange(ref numOutstandingThreadRequests, count + 1, count);</pre>					
439	<pre>if (prev == count)</pre>					
440	(
441	ThreadPool.RequestWorkerThread();					
442	break;					
443	}					
444	count = prev;					
445	}					

One epol1 thread may not be queuing things fast enough and causing EnsureThreadRequested to go down the slow path. Based on the time spent under that method most of the time is spent under RequestWorkerThread, which is the slow path where at least one thread had not found work in the thread pool. Hence the time spent spin-waiting for a short time in CLRLifoSemaphore::Wait, which means for about 20% of the time a thread pool thread is running out of work for a very short duration.

There is also a possibility the epol1 thread is getting starved a bit by the spin-waiting, though if that is removed it would likely translate into higher context-switch CPU time. complus_ThreadPool_UnfairSemaphoreSpinLimit=0 disables the spin-waiting there, may be interesting to try, to see if it noticeably affects the time spent under EnsureThreadRequested on the epol1 thread. If there is an effect then maybe it could use an earlier sleep.

https://github.com/dotnet/runtime/issues/33669#issuecomment-601459220

Which started a discussion



Without it, there are on average 62 threads in the thread pool (Environment.ProcessorCount returns 28) and as we can see in the histogram below, more than 10 of them are almost never busy:

https://github.com/dotnet/runtime/issues/33669#issuecomment-601675592

#20518: Is it possible to tune request parsing any further?

Member ····



If I remove it, I get something around 40-50k RPS gain.

@GrabYourPitchforks is there any chance that you could take a look at the parsing logic and see if there are any possibilities to optimize it any further? I know that you have a LOT of expertise in the low-level tuning of text operations.

I've prepared a copy of the TE logic and encapsulated it into a benchmark that can be run by doing the following:

git clone https://github.com/adamsitnik/aspnetcore.git parsing cd parsing git checkout techEmpowerParsing ./build.sh ./.dotnet/dotnet run -c Release -f netcoreapp5.0 --filter TechEmpowerHttpParserBenchmark --project ./src/Servers/Kestrel/p

halter73 commented on 3 Apr

Member ····

HTTP/1.x parsing has already been pretty heavily optimized by people like @benaadams.

We don't really do text operations at the HttpParser-level. We treat everything as bytes. Instead we use ReadOnlySpan.IndexOf (or ReadOnlySequence.PositionOf for headers that span multiple blocks), to search for the next \n byte in the input stream and slice. These methods have been vectorized where possible.

There's some validation that happens to verify that the headers don't contain any invalid bytes (also vectorized), and there's some copying that happens for headers that span multiple blocks, but there's no decoding, string allocations or anything like that going on in the platform benchmarks. It just calls the no-op BenchmarkApplication.public.OnHeader(ReadOnlySpan
byte> name, ReadOnlySpan<byte> value) implemenation.

adamsitnik commented on 6 Apr

Member Author ···

I just looked at the current implementation and my first thought is: why do we iterate over a parser line so many times? And why do we use vectorized methods? Vectorization has an overhead and for small inputs, it very often means even worse performance. Are typical HTTP header names and values long? (sorry I am a web n00b).

https://github.com/dotnet/aspnetcore/issues/20518

#20885: Make HTTP/1.1 startline parsing "safe"



pr-benchmarks (bot) commented on 22 Apr ··· · ·: **Baseline** Starting baseline run on 'f9a9788c67355351f6c2844489b71be495c48953'... RequestsPerSecond: 341,134 Max CPU (%): 99 WorkingSet (MB): 88 Avg. Latency (ms): 6.79 Startup (ms): 508 First Request (ms): 156.07 Latency (ms): 0.5 Total Requests: 5,149,335 Duration: (ms) 15,090 Socket Errors: 0 Bad Responses: 0 Build Time (ms): 24,012 Published Size (KB): 120,355 SDK: 5.0.100-preview.2.20120.3 5.0.0-preview.4.20220.19 Runtime: ASP.NET Core: 5.0.0-preview.5.20221.4 PR

Starting PR run on 'd7	o5e580a05ce1c830	10d7426d2df4f8cb3	e6430'			
Description RP	5 CPU (%) Mer	nory (MB) Avg.	Latency (ms) Startu	up (ms) Bui	ld Time (ms) P	ublished Size (KB)
Before <mark>341,13</mark>	4 99	88	6.79	508	24012	120355
After 350,22	8 99	87	6.62	497	7504	120355

https://github.com/dotnet/aspnetcore/pull/20885

#35330: Parallelize epoll events on thread pool and process events in the same thread

Conversation 93 -O- Commits 13 E Checks 116 ± Files changed 2 kouvel commented on 23 Apr Member 😧 … It was seen that on larger machines when fewer epoll threads are used, throughput drops significantly. An issue was that the epoll threads were not able to queue work items to the thread pool quickly enough to keep thread pool threads fully occupied. When the thread pool is not fully occupied, thread pool threads end up waiting for work and enqueues are much slower as they need to release a thread, creating a positive feedback loop, and lots of thread pool threads being released to look for work items and not finding any. It also doesn't help that the thread pool requests many more threads than necessary for the number of work items enqueued, and that the enqueue overhead is repeated for each epoll socket event. Following @adamsitnik's idea on batching the engueues to the thread pool and requesting only one thread to limit the overhead, this change tries to reduce and delegate the overhead in epoll threads into the thread pool and automatically parallelizing that work, and to decrease the number of redundant thread pool work items a bit. • The epoll thread enqueues socket events to a concurrent queue specific to the epoll thread. If the queue is busy enough, enqueues would not contend. The epoll thread then schedules a work item to the thread pool to process events, if a work item is not already scheduled • When the work item runs, it dequeues an event, schedules another work item if necessary to parallelize the work, processes the event, and continues until the event queue is empty At most one work item is scheduled to the thread pool at a time to avoid over-parallelizing the work Since socket events are now processed on a thread pool thread already, the change also avoids scheduling a redundant thread pool work item to perform the socket operations and user callbacks • The change is relatively more beneficial when fewer epoll threads are used. A heuristic for that is being discussed separately, for now this change doesn't change the number of epoll threads. 🏂 1 🔍 12 🚀 1

Big wins!



kouvel commented on 1 May

Updated numbers below with preview 5 SDK. These are with hill climbing disabled.

JsonPlatform

28-proc x64 machine

512 connections	Epoll threads	Before	After	Diff
	16	937492	983232	4.9%
	4	1054384	1095836	3.9%
	2	1004742	1136945	13.2%
	1	717291	<mark>1175142</mark>	63.8%
Max		1054384	1175142	11.5%

12-proc x64 machine

512 connections	Epoll threads	Before	After	Diff
	16	462025	502741	8.8%
	4	486645	536467	10.2%
	2	509969	568554	11.5%
	1	525168	586676	11.7%
Max		525168	586676	11.7%

FortunesPlatform

This benchmark seems to be affected by the number of connections and epoll threads. On the x64 machines, in some cases with 512 connections and 1 epoll thread the change seems to be performing slightly worse than the baseline, while with 256 connections and 1 epoll thread the change seems to be performing slightly better.

28-proc x64 machine

256 connections	Epoll threads	Before	After	Diff
	8	295163	303885	3.0%
	4	308736	314709	1.9%
	2	319814	324905	1.6%
	1	322504	334484	<mark>3.7%</mark>
Max		322504	334484	3.7%

512 connections	Epoll threads	Before	After	Diff
	16	295230	306175	3.7%
	4	313131	303398	-3.1%
	2	320548	311698	-2.8%
	1	326749	314887	-3.6%
Max		326749	314887	-3.6%

https://github.com/dotnet/runtime/pull/35330#issuecomment-622306651

2nd PR: Single epoll thread per 28 cores

⊱ Merged	Single Chang	e epoll thread per 28 cores #35800 tes from all commits - File filter Jump to 2 files viewed (i)
✓ 111	sr	c/libraries/System.Net.Sockets/sockets/SocketAsyncEngine.Unix.cs
59	+	private static int GetEngineCount()
60	+	{
61	+	// The responsibility of SocketAsyncEngine is to get notifications from epoll kqueue
62	+	// and schedule corresponding work items to ThreadPool (socket reads and writes).
63	+	1/
64	+	// Using TechEmpower benchmarks that generate a LOT of SMALL socket reads and writes under a VERY HIGH load
65	+	// we have observed that a single engine is capable of keeping busy up to thirty x64 and eight ARM64 CPU Cores.
66	+	//
67	+	// The vast majority of real-life scenarios is never going to generate such a huge load (hundreds of thousands of requests per second)
68	+	// and having a single producer should be almost always enough.
69	+	//
70	+	// We want to be sure that we can handle extreme loads and that's why we have decided to use these values.
71	+	11
72	+	// It's impossible to predict all possible scenarios so we have added a possibility to configure this value using environment variables.
73	+	<pre>if (uint.TryParse(Environment.GetEnvironmentVariable("DOTNET_SYSTEM_NET_SOCKETS_THREAD_COUNT"), out uint count))</pre>
74	+	{
75	+	return (int)count;
76	+	}
77	+	
78	+	Architecture architecture = RuntimeInformation.ProcessArchitecture;
79	+	<pre>int coresPerEngine = architecture == Architecture.Arm64 architecture == Architecture.Arm</pre>
80	+	? 8
81	+	: 30;
82	+	
83	+	<pre>return Math.Max(1, (int)Math.Round(Environment.ProcessorCount / (double)coresPerEngine));</pre>

https://github.com/dotnet/runtime/pull/35800/

How to read the results



 adamsitnik commented on 6 May • edited →
 Member

 How to read the results
 G
 H
 I
 J

 Image: Second second

1	Machine	Connections	Benchmark	before #35330) (#3	5330	1ET CD	1ET LD	2ET CD	2ET LD	4ET CD	4ET LD	MAX
2	Citrine 28 cores		128 PlaintextPlatform	7,274,914	7,385	,508	7,656,160	7,672,906	7,396,472	7,431,925	7,371,931	7,361,014	7,672,906
3			JsonPlatform	728,738	753	,185	836,097	850,134	787,839	771,172	753,278	756,862	850,134
4			FortunesPlatform	288,242	293	,637	300,804	303,350	305,006	303,894	285,718	286,513	305,006
5			Fortunes Batching	169,087	167	,237	179,321	174,056	171,451	172,060	168,792	170,285	179,321

Author

··· ··

before #35330 means results before merging #35330

#35330 means code after merging #35330

xET yD means code after merging #35330 with the micro-optimizations from this PR, using x epoll threads, using y Dictionary. y: c stands for Concurrent while L for generic dictionary used under Lock. So 1ET cD means single epoll thread using Concurrent Dictionary.

Fortunes Batching means Fortunes Platform benchmark executed with a copy of Npgsql.dll provided by @roji that implements batching

Colors: default MS Excel color scheme where red means the worst and green means the best result.

x64 12 Cores (the perf machine)

Let's start with something simple:

	А	В	С	D	E	F	G	Н	1	J	К	L
1 M	lachine	Connections	Benchmark	before #35330	#35330	1ET CD	1ET LD	2ET CD	2ET LD	4ET CD	4ET LD	MAX
28 P	erf 12 cores	12	8 PlaintextPlatform	4,548,601	4,581,534	4,660,689	4,616,641	4,661,761	4,632,020	4,565,981	4,598,769	4,661,761
29			JsonPlatform	438,914	456,929	504,433	498,451	486,133	488,477	461,018	463,563	504,433
30			FortunesPlatform	120,766	127,799	138,110	137,034	133,648	135,273	129,064	128,391	138,110
31												
32		25	6 PlaintextPlatform	4,520,799	4,728,698	5,216,799	5,161,151	5,136,942	5,140,176	4,961,895	4,959,549	5,216,799
33			JsonPlatform	441,074	464,803	541,968	542,939	524,208	521,502	500,784	497,616	542,939
34			FortunesPlatform	123,775	132,081	139,893	140,190	135,937	135,015	131,856	131,496	140,190
35												
36		51	2 PlaintextPlatform	4,439,709	4,915,243	5,518,636	5,407,579	5,364,027	5,324,880	5,132,818	5,075,158	5,518,636
37			JsonPlatform	456,198	480,191	556,005	558,511	551,288	550,309	514,124	513,776	558,511
38			FortunesPlatform	121,289	130,383	128,165	128,639	128,783	128,387	130,741	130,427	130,741
39												
40		1,02	4 PlaintextPlatform	4,270,802	4,856,757	5,251,659	5,201,533	5,239,870	5,234,974	4,993,844	4,976,044	5,251,659
41			JsonPlatform	453,737	480,158	571,392	567,129	550,002	557,413	514,228	512,937	571,392
42			FortunesPlatform	108,143	118,506	123,303	124,303	122,504	121,022	119,493	119,980	124,303
43												
44		20,00	0 PlaintextPlatform	3,886,569	3,960,775	4,065,620	4,274,768	4,384,982	4,276,343	4,075,886	3,941,157	4,384,982
45			JsonPlatform	309,933	333,290	382,610	380,935	365,846	373,034	345,486	346,800	382,610
46			FortunesPlatform	94,303	105,309	110,314	109,566	110,206	109,578	107,308	106,063	110,314

As we can see, switching to a single epoll thread and using ConcurrentDictionary gives the best results - the 1ET CD column is the greenest one. No regressions, pure win.

There are two cases where having more epoll threads gives better results:

- JsonPlatform using 512 connections. We could get 130k instead of 128k. The difference is so small that it's ignorable
- PlaintextPlatform using 20_000 connections. The difference is small, but IMHO Plaintext is the most artificial benchmark (because of the pipelining and super small response) and making the heuristic more complex to get few extra % here is not worth it.

x64 28 Cores (Citrine, the TechEmpower machine)

TechEmpower hardware:

	A	В	С	D	E	F	G	н	1	J	K	L
1	Machine	Connections	Benchmark	before #35330	#35330	1ET CD	1ET LD	2ET CD	2ET LD	4ET CD	4ET LD	MAX
2	Citrine 28 cores	12	8 PlaintextPlatform	7,274,914	7,389,508	7,656,160	7,672,906	7,396,472	7,431,925	7,371,931	7,361,014	7,672,906
3			JsonPlatform	728,738	753,185	836,097	850,134	787,839	771,172	753,278	756,862	850,134
4			FortunesPlatform	288,242	293,637	300,804	303,350	305,006	303,894	285,718	286,513	305,006
5			Fortunes Batching	169,087	167,237	179,321	174,056	171,451	172,060	168,792	170,285	179,321
6								111				
7		25	6 PlaintextPlatform	8,855,217	8,898,258	8,997,581	8,968,754	8,627,402	8,595,274	8,673,520	8,686,706	8,997,581
8			JsonPlatform	941,176	952,582	1,077,994	1,092,889	1,006,786	996,792	981,305	982,572	1,092,889
9			FortunesPlatform	291,339	301,213	339,556	332,482	321,783	319,000	316,409	315,030	339,556
10			Fortunes Batching	311,945	299,331	343,612	333,443	304,352	307,452	294,810	290,910	343,612
11												
12		51	2 PlaintextPlatform	8,785,644	9,139,882	9,327,239	9,266,524	9,153,825	9,153,160	9,200,770	9,139,853	9,327,239
13			JsonPlatform	919,425	956,259	1,123,093	1,112,853	1,058,435	1,074,620	1,044,485	1,071,394	1,123,093
14			FortunesPlatform	289,177	302,984	311,268	309,895	318,266	314,895	318,296	315,729	318,296
15	8		Fortunes Batching	358,163	349,256	407,231	405,158	388,114	384,782	367,437	368,849	407,231
16												
17		1,02	4 PlaintextPlatform	8,798,429	9,093,448	9,266,961	9,275,661	9,305,794	9,204,526	9,373,390	9,310,366	9,373,390
18			JsonPlatform	917,482	983,014	1,132,661	1,143,559	1,084,105	1,086,495	1,074,888	1,069,799	1,143,559
19			FortunesPlatform	261,790	273,522	301,019	296,848	294,755	296,556	292,395	293,485	301,019
20			Fortunes Batching	372,989	374,679	417,499	419,579	405,383	412,505	400,590	398,749	419,579
21												
22		20,00	0 PlaintextPlatform	6,711,039	6,707,423	7,170,514	7,074,713	7,101,042	7,137,157	7,056,471	7,150,295	7,170,514
23	/		JsonPlatform	742,247	754,620	723,171	764,932	824,890	838,540	827,230	823,884	838,540
24			FortunesPlatform	208,385	220,029	251,166	241,759	243,700	242,205	239,267	238,290	251,166
25			Fortunes Batching	289,530	301,026	326,969	337,845	328,056	321,860	306,706	326,969	337,845

Again, switching to a single epoll thread and using ConcurrentDictionary gives the best results - the **IET CD** column is the greenest one.

There are few cases where having more epoll threads gives better results:

- small and ignorable differences within the marigin of error:
 - 300k vs 305k for Fortunes using 128 connections
 - 311k vs 318k for Fortunes using 512 connections
 - 9268k vs 9373k for Plaintext using 1024 connections
- a regression from 742k to 723k for JsonPlatform with 20_000 connections. It's a 2.5% regression, so it's small and the two other benchmarks (Plaintext and Fortunes) give the best results for this config so I think that it's acceptable

Very good thing: the throughput of JSON and Fortunes benchmarks rise when the number of clients increases (to some point ofc). We did not have that before.

Another great thing: 417,499 for Fortunes 1024 connections with latest bits from @roji It's top 10 of Fortunes ;)

x64 56 Cores (Mono machine)

	А	В	С	D	Е	F	G	Н	I	J	К	L
1	Machine	Connections	Benchmark	before #35330	#35330	1ET CD	1ET LD	2ET CD	2ET LD	4ET CD	4ET LD	MAX
70	Mono 56 cores	128	PlaintextPlatform	6,011,013	6,508,597	6,964,004	6,767,192	6,523,399	6,478,501	6,489,369	6,542,138	6,964,004
71			JsonPlatform	462,300	673,968	664,928	639,536	660,309	669,500	585,216	670,094	673,968
72												
73		256	PlaintextPlatform	6,896,236	6,906,699	6,908,931	6,899,911	6,928,565	6,923,914	6,925,540	6,915,618	6,928,565
74			JsonPlatform	600,973	980,908	922,999	908,916	1,038,827	957,700	995,169	997,400	1,038,827
75												
76		512	PlaintextPlatform	6,941,870	6,941,820	6,922,900	6,926,490	6,950,605	6,952,889	6,954,533	6,953,451	6,954,533
77			JsonPlatform	623,578	1,079,661	1,042,180	1,038,941	1,175,365	1,118,234	1,132,995	1,097,764	1,175,365
78												
79		1,024	PlaintextPlatform	6,960,810	6,962,596	6,935,703	6,949,966	6,960,988	6,961,073	6,964,004	6,959,078	6,964,004
80			JsonPlatform	741,710	1,138,508	982,306	1,048,731	1,191,322	1,206,231	1,145,028	1,175,588	1,206,231
81												
82		20,000	PlaintextPlatform	6,825,034	6,784,191	6,730,037	6,750,541	6,863,600	6,847,910	6,830,042	6,855,341	6,863,600
83			JsonPlatform	660,291	919,557	657,186	728,571	949,984	974,414	954,127	961,749	974,414

With 56 cores having a single epoll thread is not enough. Having two gives us the most optimal solution that is improving all cases.

There are two cases where having more epoll threads gives better results, but all of them are small and ignorable differences within the margin of error:

- 6954k vs 6950k for Plaintext using 256 connections
- 6964k vs 6960k for Plaintext using 512 connections

There are two where having less epoll threads gives better results:

- ignorable 660k vs 673k for JsonPlatform using 128 connections
- 6523k vs 6964k for PlaintextPlatform using 128 connections. Having a single epoll thread could give us better results, but we still have an improvement compared to base 6011k. We could reach it by setting the MinHandles to 128 instead of 32, but I don't think that it's worth it it's rather unlikely that such a beefy machine is going to be used for handling such a small load.

Very nice thing: the gains are really big. Even up to x2 for Json with 512 connections.

The Fortunes benchmark is not included because for some reason this machine can not currently access the db server.

ARM64 32 Cores

Here is where things get complicated:

	A	В	С	D	E	F	G	н	I	J	к	L
1 Machine		Connections	Benchmark	before #35330	#35330	1ET CD	1ET LD	2ET CD	2ET LD	4ET CD	4ET LD	MAX
49 ARM 32 c	cores	128	PlaintextPlatform	5,325,320	5,248,309	3,561,284	3,845,438	4,781,959	4,898,041	5,358,677	5,298,232	5,358,677
50			JsonPlatform	470,719	467,996	370,726	360,950	426,570	464,457	437,481	446,316	470,719
51			FortunesPlatform	70,159	79,601	68,324	57,458	65,881	75,468	87,091	74,375	87,091
52												
53		256	PlaintextPlatform	5,443,043	5,433,406	4,605,492	4,299,638	5,592,394	5,347,411	5,599,302	5,662,091	5,662,091
54			JsonPlatform	455,767	420,229	372,432	377,698	425,805	462,432	455,476	453,779	462,432
55			FortunesPlatform	73,379	76,414	73,532	72,876	85,140	82,368	74,248	68,891	85,140
56												
57		512	PlaintextPlatform	5,143,935	5,644,389	5,017,068	4,451,933	5,453,011	5,431,783	5,937,616	5,839,524	5,937,616
58			JsonPlatform	425,086	397,756	385,939	370,699	459,059	460,038	426,664	451,730	460,038
59			FortunesPlatform	80,027	79,361	51,971	60,200	75,948	64,618	86,416	78,160	86,416
60												
61		1,024	PlaintextPlatform	5,289,294	5,409,985	5,485,081	4,565,115	5,495,414	5,348,224	5,833,511	5,913,468	5,913,468
62			JsonPlatform	350,471	376,589	345,595	395,434	467,338	446,565	432,101	442,890	467,338
63			FortunesPlatform	59,300	53,292	49,349	49,958	61,679	60,924	54,847	55,797	61,679
64												
65		20,000	PlaintextPlatform	3,799,859	4,109,911	4,589,044	4,522,555	4,606,430	4,584,374	4,826,834	4,478,308	4,826,834
66			JsonPlatform	246,717	258,675	294,565	316,978	289,399	300,353	358,134	299,867	358,134
67			FortunesPlatform	44,415	36,242	32,431	32,831	61,568	49,223	55,955	46,665	61,568

Having a single epoll thread, no matter what dictionary we use gives us a lot of red color (except the case with 20k connections).

There is no obvious dependency between the number of connections and the number of threads (like the more connections the more threads we need). If we take a look at the numbers before our changes it looks like this machine is struggling to scale up when the number of connections grows (JSON numbers are: 470->455->425->350->246). This requires an independent investigation.

Using 4 epoll threads gives us more improvements than using two. There is only one regression: JSON using 128 connections. Again, I think that for this number of Cores we should optimize for many connections and I hope that this is acceptable.

#36371: Try using socket syscalls that accepts a single buffer to improve performance

୍ଲ ପ୍ <u>ର</u> (Q Conversation 19 -∞ Commits 7 E Checks 130 ± Files changed 10		A	В	С	D	E	F
		1	Machine	Connections	Benchmark	before	after	ratio
9	tmds commented on 13 May	Member 2	Citrine 28 cores	512	PlaintextPlatform	9,311,240	9,358,872	0.51%
		3			JsonPlatform	1,149,483	1,180,958	2.74%
	This is for benchmarking to see if using syscalls that accept a single buffer has a measurable impact on performance	ce. 4			FortunesPlatform	311,110	318,603	2.41%
		5			Fortunes Batching	418,096	415,193	-0.69%
		6						
KO O		7						
recy	msg -> recv	8	Perf 12 cores	512	PlaintextPlatform	5,750,630	5,888,939	2.41%
sen	dmsg -> send	9			JsonPlatform	553,999	575,528	3.89%
		10			FortunesPlatform	127, <mark>1</mark> 90	130,225	2.39%
		11						
		12						
		13	Mono 56 cores	512	PlaintextPlatform	6,948,232	6,934,850	-0.19%
		14			JsonPlatform	1,177,622	1,162,769	-1.26%
		16						
		17	AMD 46 cores	512	JsonPlatform	667,898	670,767	0.43%
		18			FortunesPlatform	240,173	262,262	9.20%

#36635: Is it possible to optimize JSON serialization any further?

randly, one of them can be JSON serialization itself. thousand requests per second. where the could you please take a look at the breakdown below and see if there is anything that we could improve? the kdown we are spending the take a look at the breakdown below and see if there is anything that we could improve? The take at the take at tak	rank of the mean o	all, one of them can be JSON serialization itself.	recent changes in the networking stack allowed us to improve the throughput by +20%. But we a ere we won't be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further and we are looking for some other places that could be able to optimize it any further an	re slowly getting to the point
Trail, one of them can be JSON serialization itself.	rail, one of them can be JSON senalization itself.	ally, one of them can be JSUN senaitzation itself.		a be improved.
<text><text><text></text></text></text>	<text><text><text></text></text></text>	today, we are spending + - 4.6% of the total CPU time for the JSON serialization in the JSON benchmark. 1% of CPU time are started to the total of the total control time are started to	urally, one of them can be JSON serialization itself.	
<text></text>	<text></text>	Are to the two has been and be	of today, we are spending +- 4.6% of the total CPU time for the JSON serialization in the JSON ben	nchmark. 1% of CPU time
			islates to circa 10 thousand requests per second.	
		Set	teveharter Could you please take a look at the breakdown below and see if there is anything that v	we could improve?
		Provide state	· breakdown:	
		Characterize the state st	,	Inc \$67 Inc 7 Inc Awy 7 Inc Cl 7 Eac \$67 Eac 7
		Or both the base base base base base base base bas	* Effertunden brackelfartereiten bracksbepterion: Handelepartein kanten gestern Mennas (byten Baller) (separce't cantis is kon) * Effertunden bracksbeptereiten bracksbezohnsklepketon son vielen gestern bestern be	100.025.670.0 1.0 25.670 0.0 0 100.025.670.0 1.0 25.670 4.3 1.116
		Image: Section 1	 VErytem flock and replaces that has been been been been been been to be the base of the b	76.3[12,667.0] 1.0] 19,667 0.1 37 75.1[12,209.0] 1.0] 19,209 1.6 423
			11 Bioseck and Coll Archivelype 12 Bioseck and Coll Coll Archivelype 13 Bioseck and Coll Coll Archivelype 14 Bioseck and Coll Coll Archivelype 15 Bioseck and Coll Coll Archivelype 16 System Set Analysis Bioseck and Coll Coll Coll Coll Coll Coll Coll Col	0.3 119.0 1.0 119 0.4 95 0.4 103.0 1.0 103 0.3 82
			1 ¹¹ Systems. Net A low Clystems. Net Jack With With Stack Institution Particles Particular Systems Rev Low Linder Stark Of Const. Doil 1 ¹⁴ Systems. Net Jack With Stack Institution With Const. Stark End Linder Stark Institution Const. Stark Million With Stack 20 1 ¹⁴ Systems. Net Jack With Stack Institution With Const. Stark End Linder Stark Institution Const. Stark With Stack 20 1 ¹⁴ Systems. Net Jack With Stack Institution With Stack 20 1 ¹⁴ Stark With Stark With Stack 20 1 ¹⁴ Stark With Stark 20 1 ¹⁴	0.3 85.0 1.0 85 0.2 59 0.7 36.0 1.0 36 0.1 30
			Tametaliyeetar (eos, eo Tametaliyeetar (eos, eos) Tametaliyeetar (eos) Tametaliyeetar (eo	0.0 11.0 1.0 11 0.0 0
	• • • • • • • • • • • • • • • • • • •		Clyptem D PeptientSystem (D PeptientSystem) System D PeptientSystem (D PeptientSystem) System) System D Perturbation (Statistics)	58 1.485.0 1.0 1.485 0.5 122 3.4 876.0 1.0 876 2.7 683
			* Dispondu at 21 Nov * Dispondu at 21 Nov	1.3 844.0 1.0 844 0.0 7 1.5 374.0 1.0 374 1.1 279
	• • • • • • • • • • • • • • • • • • •	Control	Bernflaßingendes, form, sein Bernflaßingendes, form, sein Bernflaßingendes, form, sein	1.1 295.0 1.0 295 0.0 0
			* Byten BelowSystem Section URLowWebsCheerHelper)	0.7 187.6 1.6 167 0.6 144
	• Control Control <td></td> <td> </td> <td></td>		 	
			- Texania (and a second s	0.2 52.0 1.0 52 0.2 40
			 	0.1 36.0 1.0 36 0.1 29
			Boundstofferlang SyleMethame Constraints Second Statement	0.1 25.0 1.0 25 0.1 16
	• Second data by fast all and the fast all	Image: Second	* Boomback_B_griddbight ** Discretification of the second seco	0.7 19.0 1.0 19 0.1 16 0.7 18.0 1.0 18 0.1 14
		••••••••••••••••••••••••••••••••••••	Theorem is and increase Payel Channel Theorem is and increase in the Control of the Con	0.1 15.0 1.0 15 0.1 14
			Borowshi so Kholendar Palaki pijeken kala sine Borowshi so Kholendar Palaki pijeken kala sine	0.1 13.0 1.0 13 0.0 11
			 	0.0 10.0 1.0 10 9
			etectrics.scs.v.cecorevisation	0.0 2.0 1.0 5 0.0 6 0.0 8.0 1.0 8 0.0 7
			*2 Torond schuldstatudicaptivetie *2 Torond schuldstatudicaptivetie // Torond schuldstatudicaptivetie	04 64 14 6 04 6 04 55 14 5 04 5
			*Bipconth schildon MethodFunne App	6 B.0 B. B.0 B.0
The Second Secon				
		press from June 2	and the second s	
(gran. Nat. Juni. gran. Juni. gran. Juni.		prime from 2 continues and a second	and the second se	
The first first product of the first first product of the first product			System, Text, Jacol.	
ente de la bechan han benere apple a gran de la benere apple de la construction de la benere de	palae. Nat Jane Jane Jane Jane Jane Jane Jane Jane	system met, John System, Fext, John Sersal Blon,	gytes, feet, Janet Sytes, feet, Janet, Seefall 1910s, Janetaseverant L. Spitter, Test, Janet,	
preserved and an and a second se	tes.TexT.JuniTyytes.TexT.Jun.Striplization.Converters.05pttbffabl/Converter.102ptforeManchasks.Junkessapp110hfpbttbc(Lass.	antini, Tent, Jacob Jacob Speet y Sale 1, 2000 System, Tent, Jacob Jacob Speet Antini, S	geter, Text, Jion System, Text, Jion, JionProperty Life L. System, Text, Jion System, Jion System, Text, Jio	
ten Tert Son Sector (alla Sector for Sector Se		teer/heit/jiel/system.fret/jiel/set/set/set/set/set/set/set/set/set/set	Test Test Test Test Test Test Test Test	
	a. Text, Son System, Text, Son, Serval Latter and Text an	. Tert. Juni System. Tert. Jone. Serialization. June Converter: [[FlatfereBenchemtsJuneRess.ge]]; (System. Tert. June. June (Lines. Lines.		
en. Text. Joon System. Text. Joon. Serialization. JoonGoverter' ([Platformbechavits. Joor/essage)): ariteCore(class. System. Text. Joon.Off/Disoneriter, 10, class.	a. retr. Annihytes retr. Jasse and retr. Jasse and retre and	. htt. heidigsten, itst. Jone Serialization. Jandometer: "[Flatteredendenden: Jandessage]] :Typitit(class. System, Ret. Jano, Heidigsten,	history rest. Journey stem rest. Journey of a statistic on the statistic of the statistic o	

Member 🕤 … stephentoub commented on 19 May If we make the simplifying assumptions that Utf8JsonWriter is as optimized as it can be and that JsonSerializer must use it for its JSON writing, there's still some measurable overhead that could be reduced, but it's not the majority: using BenchmarkDotNet.Attributes; using BenchmarkDotNet.Running; using System.Buffers; using System.Text.Json; [MemoryDiagnoser] public class Program static void Main(string[] args) => BenchmarkSwitcher.FromAssemblies(new[] { typeof(Program).Assembly }).Run(args); private static readonly JsonSerializerOptions SerializerOptions = new JsonSerializerOptions(); private static readonly ArrayBufferWriter<byte> Writer = new ArrayBufferWriter<byte>(); public struct JsonMessage { public string message { get; set; } [Benchmark] public void Serialize() { Writer.Clear(); using (var utf8JsonWriter = new Utf8JsonWriter(Writer)) JsonSerializer.Serialize(utf8JsonWriter, new JsonMessage { message = "Hello, World!" }, SerializerOptions); } [Benchmark] public void Write() { Writer.Clear(): using (var utf8JsonWriter = new Utf8JsonWriter(Writer)) £ var message = new JsonMessage { message = "Hello, World!" }; utf8JsonWriter.WriteStartObject(); utf8JsonWriter.WriteString("message", message.message); utf8JsonWriter.WriteEndObject(); } } Error StdDev Gen 0 Gen 1 Gen 2 Allocated Method Mean Serialize 199.1 ns 0.68 ns 0.60 ns 0.0229 144 B Write 126.0 ns 1.88 ns 1.67 ns 0.0191 120 B

https://github.com/dotnet/runtime/issues/36635

#1519: try Suggestion from Stephen and use Write methods directly

```
using (Utf8JsonWriter utf8JsonWriter = new Utf8JsonWriter(writer.Output))
```

https://github.com/aspnet/Benchmarks/pull/1519

}

```
JsonSerializer.Serialize<JsonMessage>(utf8JsonWriter, new JsonMessage { message = "Hello, World!" }, SerializerOptions);
var message = new JsonMessage { message = "Hello, World!" };
utf8JsonWriter.WriteStartObject();
utf8JsonWriter.WriteString("message", message.message);
utf8JsonWriter.WriteEndObject();
```

@stephentoub +20k in JSON!

Before:

RequestsPerSecond:	1,149,856
Max CPU (%):	99
WorkingSet (MB):	407
Avg. Latency (ms):	1
Startup (ms):	196
First Request (ms):	31.96
Latency (ms):	0.12
Total Requests:	17,362,051
Duration: (ms)	15,100
Socket Errors:	0
Bad Responses:	0
Build Time (ms):	4,001
Published Size (KB):	102,238
SDK:	5.0.100-preview.5.20264.2
Runtime:	5.0.0-preview.6.20262.14
ASP.NET Core:	5.0.0-preview.5.20255.6

After:

RequestsPerSecond:	1,171,304
Max CPU (%):	100
WorkingSet (MB):	410
Avg. Latency (ms):	0.86
Startup (ms):	202
First Request (ms):	31.5
Latency (ms):	0.1
Total Requests:	17,686,603
Duration: (ms)	15,100
Socket Errors:	0
Bad Responses:	0
Build Time (ms):	4,001
Published Size (KB):	102,238
SDK:	5.0.100-preview.5.20264.2
Runtime:	5.0.0-preview.6.20262.14

With dotnet/runtime#36371

RequestsPerSecond:	1,203,964
--------------------	-----------



stephentoub commented on 19 May

Does this violate the TE spec? "A JSON serializer must be used to convert the object to JSON."

adamsitnik commented on 19 May

"A JSON serializer must be used to convert the object to JSON."

It does. 1200k looks tempting, but I am afraid I should close this issue.



adamsitnik closed this on 19 May



#1520: Cache Utf8JsonWriter



adamsitnik commented on 19 May

Another suggestion from @stephentoub

The gain is on average around +3k RPS

[ThreadStatic] private static Utf8JsonWriter t writer; private static void Json(ref BufferWriter<WriterAdapter> writer) writer.Write(_jsonPreamble); 1 +28,11 @@ private static void Json(ref BufferWriter<WriterAdapter> writer) writer.Commit(); Utf8JsonWriter utf8JsonWriter = t_writer ??= new Utf8JsonWriter(writer.Output); utf8JsonWriter.Reset(writer.Output); // Body using (Utf8JsonWriter utf8JsonWriter = new Utf8JsonWriter(writer.Output)) JsonSerializer.Serialize<JsonMessage>(utf8JsonWriter, new JsonMessage { message = "Hello, World!" }, SerializerOptions); } JsonSerializer.Serialize<JsonMessage>(utf8JsonWriter, new JsonMessage { message = "Hello, World!" }, SerializerOptions);

#1547: DB Platform benchmarks microoptimizations



https://github.com/aspnet/Benchmarks/pull/1547

#37976: Perf improvements for small or value-type POCOs



steveharter commented on 16 Jun • edited by richlander 👻 ··· ··: Member For a TechEmPower benchmark which uses a one-property struct, this shows a ~1.2x serialization improvement during serialization. Note a value-type (struct) POCO is not common and should only be used when there are very few properties -- instead a POCO should be a reference-type (class). Fixes #36635 Summary of changes: • Add a LRU cache before the dictionary access that returns the metadata for the root type being (de)serialized. This is the biggest savings for small POCOs; larger POCOs are not affected since the dictionary access becomes insignificant compared to the rest of the work. Also this LRU helps most when there is low concurrency (few threads) or the same type is being repeatedly (de)serialized. • In internal code, pass the value types using in to avoid unnecessary copies. The more serializable properties a value-type contains, the bigger the savings. • Optimize property writes to include the quotes and colon as suggested by @Tornhoof. This also allowed for some AggressiveInlining fast-path changes to since the code path is now internal and specific to this optimization. • Other AggressiveInlining changes. Both added and removed. The crossgen size of STJ.dll is now 15K smaller (1071K to 1056K). • Property lookup for cache misses (due to case-insensitivity or different property ordering across JSON payloads for a given Type) is faster since the length is now embedded into the ulong key avoiding calls to SequenceEquals() when the length is different. This doesn't affect the TechEmPower scenarios. Other smaller misc changes.

- Click to expand for benchmarks
- Click to expand for temporary TechEmpower benchmark

Many JSON microbenchmarks have improved!



https://pvscmdupload.blob.core.windows.net/reports/allTestHistory%2frefs%2fheads%2fmaster_x64_Window s%2010.0.18362%2fSystem.Text.Json.Serialization.Tests.WriteJson(Location).SerializeToStream.html

June 2020: we have met the goals!





#2933: Multiplexing



roji commented on <u>31 May</u> • edited by Brar 👻

Member 😥 🚥

OK, this is finally in a state where I think it's OK to review and hopefully merge soon.

- Most of the comments in #2852 are still valid, so it's probably a good idea to look at them first.
- While #2852 was unsafe in various ways, I've done a lot of work around safety, and hopefully haven't missed anything important.
- I really recommend filtering by commit when reviewing. There are some commits before multiplexing which simply rewrite the pool using Channels, without any lock-free code. The last commit is where multiplexing occurs.
- For multiplexing, the "entry point" is in NpgsqlCommand.ExecuteReaderAsync. If multiplexing is on, we simply enqueue to the
 pool's command channel and wait. The main bulk of the actual multiplexing write logic is in ConnectorPool.Multiplexing.cs. The
 read logic is in NpgsqlConnector.ReadLoop. Between these three you should get a pretty good idea of what's happening.
- Note that we have a safe "over-capacity" mode; if all connections are in use (Max Pool Size), we still continue to push commands down the pipe. This is the only place which requires some basic lock-free techniques, but nothing as complex as what we used to have. Hopefully it's safe I'd appreciate a good look at this.
- Lots of tests are still lacking. For now some of the main test suites simply run twice once in multiplexing, once without. And of course a lot of manual stress testing was done with the TE Fortunes scenario.

This PR is not 100% complete - some tests are skipped, some stuff is not yet implemented (e.g. keepalive). Also, I'd rather we didn't discuss nits or refactors at this point - I think it's better to merge this and continue work in separate, self-contained PRs (this work is just too big to get done in a single PR). So be sparing with your comments if you can :)

I'm hoping we can merge this relatively quickly (is one week too aggressive? maybe two?), and release an alpha package on nuget.org to get some user testing. We're nowhere near releasing, so it's OK for this not to be completely stable yet.

Supercedes #2852



#1553: Update platform benchmarks to Npgsql 5.0.0-alpha1



roji commented on 17 Jun

And start using multiplexing

<PackageReference Include="Npgsql" Version="4.1.2" />
<PackageReference Include="Npgsql" Version="5.0.0-alpha1" />
<PackageReference Include="RedHat.AspNetCore.Server.Kestrel.Transport.Linux" Version="3.0.0-*" />

</ItemGroup>

enchmarksApps/Kestrel/PlatformBenchmarks/benchmarks.fortunes.yml 📋



enchmarkdbpass;Maximum Pool Size=256;NoResetOnClose=true;Enlist=false;Max Auto Prepare=4

enchmarkdbpass;Maximum Pool Size=256;Enlist=false;Max Auto Prepare=4;Multiplexing=true;Write Coalescing Delay Us=500;Write Coalescing Buffer Threshold Bytes=1000

https://github.com/aspnet/Benchmarks/pull/1553

Multiplexing: +59k RPS for Fortunes



Shay Rojansky 6/24 11:54 PM

BTW the Npgsql multiplexing results are finally in on the OKR dashboard. I think Seb is working on the older numbers and on a legend to explain everything, but we're at 415K RPS for Fortunes:

6 2

Fortunes Raw



5.0 vs TE 10th 415,675 356,000 (+16.8 %)

The Composite Score

⁹Scoring algorithm

The TPR scoring algorithm is intended to be fairly simple.

Goals for scoring

We have the following goals for scoring hardware performance:

- Make the scores comparable on a per-Round basis. Results from environment A should be comparable to environment B as long as both measured the implementations from the same Round (e.g., Round 19).
- Fairly easy to reproduce by hand.
- Emphasize some tests, de-emphasize others. Specifically, we want to boost the importance of Fortunes and Updates while
 decreasing the importance of Single-Query, Multi-Query, and Plaintext. Single-Query and Multi-Query are very similar and
 without reducing their importance somewhat, the performance of database querying alone would drive a large portion of
 the score. Plaintext is reduced in performance because it's the least "real-world" among our test types.

Note these goals come from both the needs of TPR hardware scoring and composite scoring for frameworks.

Semi-fixed test type biases

As a result of the goals above, we are tentatively considering the following bias coefficients per test type:

- json: 1
- single query: 0.75
- 20-query: 0.75
- fortunes: 1.5
- updates: 1.25
- plaintext: 0.75

Per-round weights

We will use the official results from our Citrine hardware environment as a "reference" environment. From these official reference results, we will:

- 1. Filter down to the TPR-tagged frameworks.
- 2. Compute an average RPS for each test type

Normalize the magnitude of each of the test types to align with the JSON test type. E.g., if the JSON average were 150,000
and the Fortunes average were 10,000, the Fortunes test would be given a normalizing coefficient of 15 (10,000 x 15 =
150,000).

4. Apply the semi-fixed biases above. Taking the Fortunes example above, the resulting weight for Fortunes would be $15 \times 1.5 = 22.5$.

These per-round weights will be rendered on the results web site, along with a link to a wiki entry (like this one) describing the scoring algorithm.

"That's insane. 50% improvement from doing nothing except upgrading .NET"



W odpowiedzi do @ben_a_adams @TFBenchmarks i@dustinmoris

That's insane. 50% improvement from doing nothing except upgrading .NET

https://twitter.com/nbevans/status/1309135751267987459

Not covered

- Multiplexing: <u>https://github.com/npgsql/npgsql/pull/2993</u>
- The Big Experiment: <u>https://github.com/tmds/Tmds.LinuxAsync/</u>
- The things that did not improve perf: AIO & io_uring:
 - <u>https://github.com/dotnet/runtime/pull/36980</u> AIO
 - <u>https://github.com/dotnet/runtime/pull/38747</u> reduce syscalls
 - <u>https://github.com/axboe/liburing/issues/97</u> io_uring
- The scenarios where the performance is far from perfect:
 - The "Mono" machine with 56 Cores: 1/3 -> 2/3
 - The AMD machine: low RPS despite powerful hardware
 - The ARM machine we don't know how our competitors perform on ARM
 - Updates benchmark +-30% time spent on waiting for a lock to be released*

Questions?

Thank You!