

Performance tricks I learned from contributing to open source .NET packages

 [@danielmarbach](https://twitter.com/danielmarbach) |  daniel.marbach@particular.net



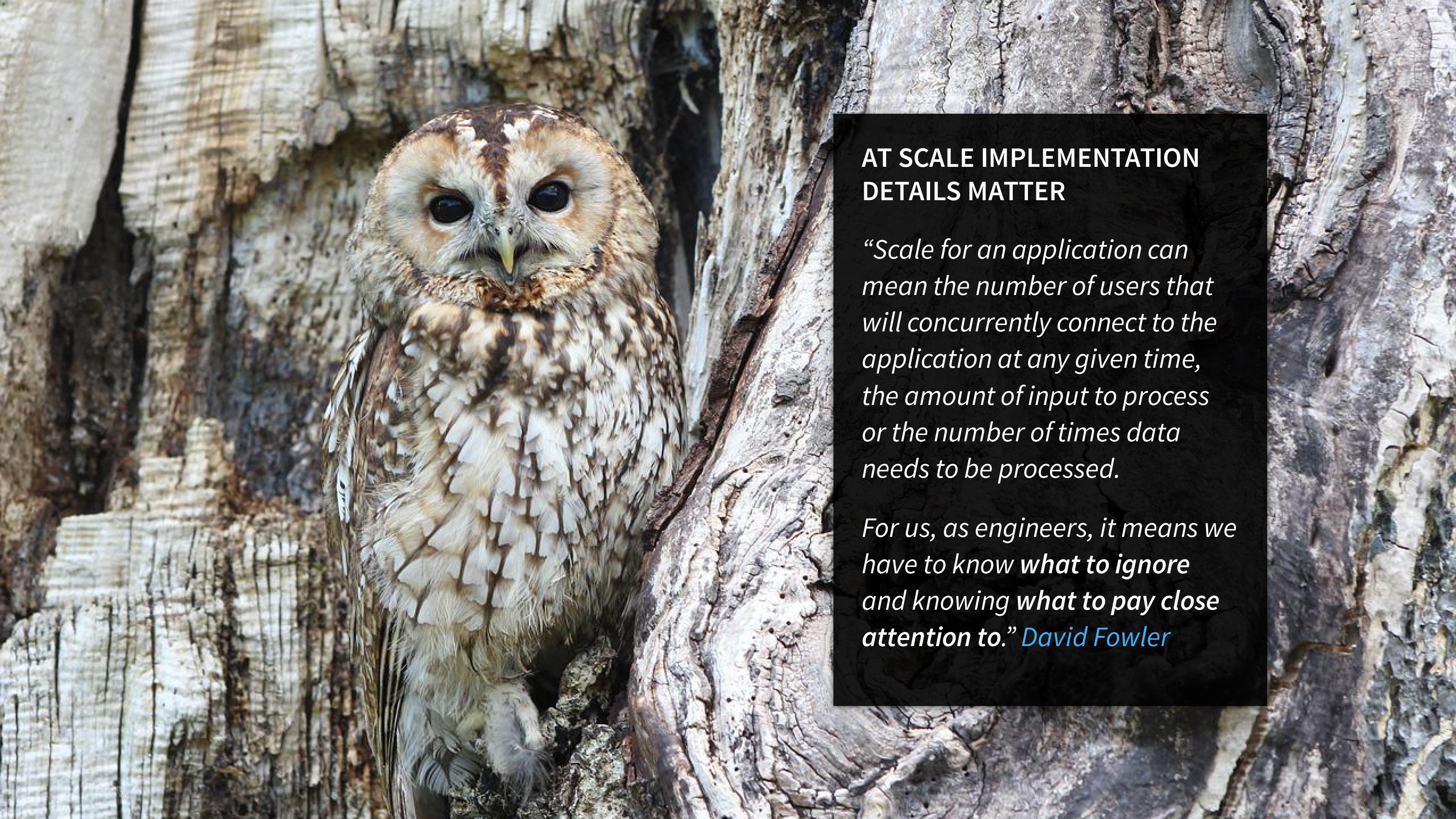
INTRODUCTION

Focus on performance optimization in .NET Code and
not architecture.



ESOTERIC

Being called out for premature optimizations.



AT SCALE IMPLEMENTATION DETAILS MATTER

“Scale for an application can mean the number of users that will concurrently connect to the application at any given time, the amount of input to process or the number of times data needs to be processed.

*For us, as engineers, it means we have to know **what to ignore** and knowing **what to pay close attention to.**” **David Fowler***

A close-up photograph of a person's hand holding a black Sharpie marker, writing the word "Rules" in a bold, black, cursive font on a light-colored wooden surface. The wood has a prominent grain and some darker knots. The hand is positioned in the lower right, with the marker tip pointing towards the bottom left. The background is blurred, showing what appears to be a garden or outdoor setting.

Rules

- Avoid excessive allocations to reduce the GC overhead
- Avoid unnecessary copying of memory

Avoid excessive allocations to reduce the GC overhead

Think at least twice before using LINQ or unnecessary
enumeration on the hot path

```
1 public class AmqpReceiver {
2
3     ConcurrentBag<Guid> _lockedMessages = new ();
4
5     public Task CompleteAsync(IEnumerable<string> lockTokens)
6         => CompleteInternalAsync(lockTokens);
7
8     Task CompleteInternalAsync(IEnumerable<string> lockTokens)
9     {
10        Guid[] lockTokenGuids = lockTokens.Select(token => new Guid(token)).ToArray();
11        if (lockTokenGuids.Any(lockToken => _lockedMessages.Contains(lockToken)))
12        {
13            // do special path accessing lockTokenGuids
14            return Task.CompletedTask;
15        }
16        // do normal path accessing lockTokenGuids
17        return Task.CompletedTask;
18    }
19 }
```

Avoid LINQ on the hot path.


```
1 public Task CompleteAsync(IEnumerable<string> lockTokens)
2   => CompleteInternalAsync(lockTokens);
3
4 Task CompleteInternalAsync(IEnumerable<string> lockTokens)
5 {
6   Guid[] array = Enumerable.ToArray(Enumerable.Select(lockTokens,
7     <>c.<>9_2_0 ??
8     (<>c.<>9_2_0 = new Func<string, Guid>(<>c.<>9.<CompleteInternalAsync>b_2_0))));
```

Avoid LINQ on the hot path.



Benchmarking Time!

We can only know the before and after when we measure it.

Method	Size	Collection	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Allocated
Before	0	Array	28.96 ns	18.868 ns	1.034 ns	1.00	0.00	0.0102	64 B
AfterV1	0	Array	17.62 ns	1.285 ns	0.070 ns	0.61	0.02	-	-
Before	0	Enumerable	29.73 ns	3.246 ns	0.178 ns	1.00	0.00	0.0102	64 B
AfterV1	0	Enumerable	23.25 ns	2.844 ns	0.156 ns	0.78	0.00	-	-
Before	0	HashSet	67.52 ns	24.479 ns	1.342 ns	1.00	0.00	0.0267	168 B
AfterV1	0	HashSet	54.60 ns	5.831 ns	0.320 ns	0.81	0.01	0.0166	104 B
Before	0	List	48.46 ns	11.809 ns	0.647 ns	1.00	0.00	0.0229	144 B
AfterV1	0	List	35.27 ns	23.899 ns	1.310 ns	0.73	0.03	0.0127	80 B
Before	1	Array	128.64 ns	72.000 ns	1.305 ns	1.00	0.00	0.0381	240 B
AfterV1	1	Array	93.27 ns	23.817 ns	1.305 ns	0.73	0.02	0.0229	144 B
Before	1	Enumerable	147.40 ns	148.138 ns	8.120 ns	1.00	0.00	0.0458	288 B
AfterV1	1	Enumerable	115.77 ns	29.583 ns	1.622 ns	0.79	0.04	0.0305	192 B
Before	1	HashSet	311.50 ns	53.412 ns	2.928 ns	1.00	0.00	0.0596	376 B
AfterV1	1	HashSet	226.89 ns	87.069 ns	4.773 ns	0.73	0.01	0.0443	280 B
Before	1	List	195.72 ns	274.095 ns	15.024 ns	1.00	0.00	0.0420	264 B
AfterV1	1	List	150.91 ns	91.677 ns	5.025 ns	0.77	0.06	0.0267	168 B
Before	4	Array	277.94 ns	41.392 ns	2.269 ns	1.00	0.00	0.0687	432 B
AfterV1	4	Array	235.52 ns	28.813 ns	1.579 ns	0.85	0.01	0.0534	336 B
Before	4	Enumerable	353.15 ns	152.507 ns	8.359 ns	1.00	0.00	0.0763	480 B
AfterV1	4	Enumerable	287.29 ns	86.240 ns	4.727 ns	0.81	0.03	0.0610	384 B

 ~20-40%
 ~20-40%



Rules



LINQ TO COLLECTION-BASED OPERATIONS

- Use `Array.Empty<T>` to represent empty arrays
- Use `Enumerable.Empty<T>` to represent empty enumerables
- Prevent collections from growing
- Use concrete collection types
- Leverage pattern matching or `Enumerable.TryGetNonEnumeratedCount`
- Wait with instantiating collections until really needed


```
1 public Task CompleteAsync(IEnumerable<string> lockTokens) {
2     IReadOnlyCollection<string> readOnlyCollection = lockTokens switch
3     {
4         IReadOnlyCollection<string> asReadOnlyCollection => asReadOnlyCollection,
5         _ => lockTokens.ToArray(),
6     };
7     return CompleteInternalAsync(readOnlyCollection);
8 }
9
10 Task CompleteInternalAsync(IReadOnlyCollection<string> lockTokens)
11 {
12     int count = lockTokens.Count;
13     Guid[] lockTokenGuids = count == 0 ? Array.Empty<Guid>() : new Guid[count];
14     int index = 0;
15     foreach (var token in lockTokens)
16     {
17         var tokenGuid = new Guid(token);
18         lockTokenGuids[index++] = tokenGuid;
19         if (_requestResponseLockedMessages.Contains(tokenGuid))
20         {
21             return Task.CompletedTask;
22         }
23     }
24     return Task.CompletedTask;
25 }
```

Avoid LINQ on the hot path.



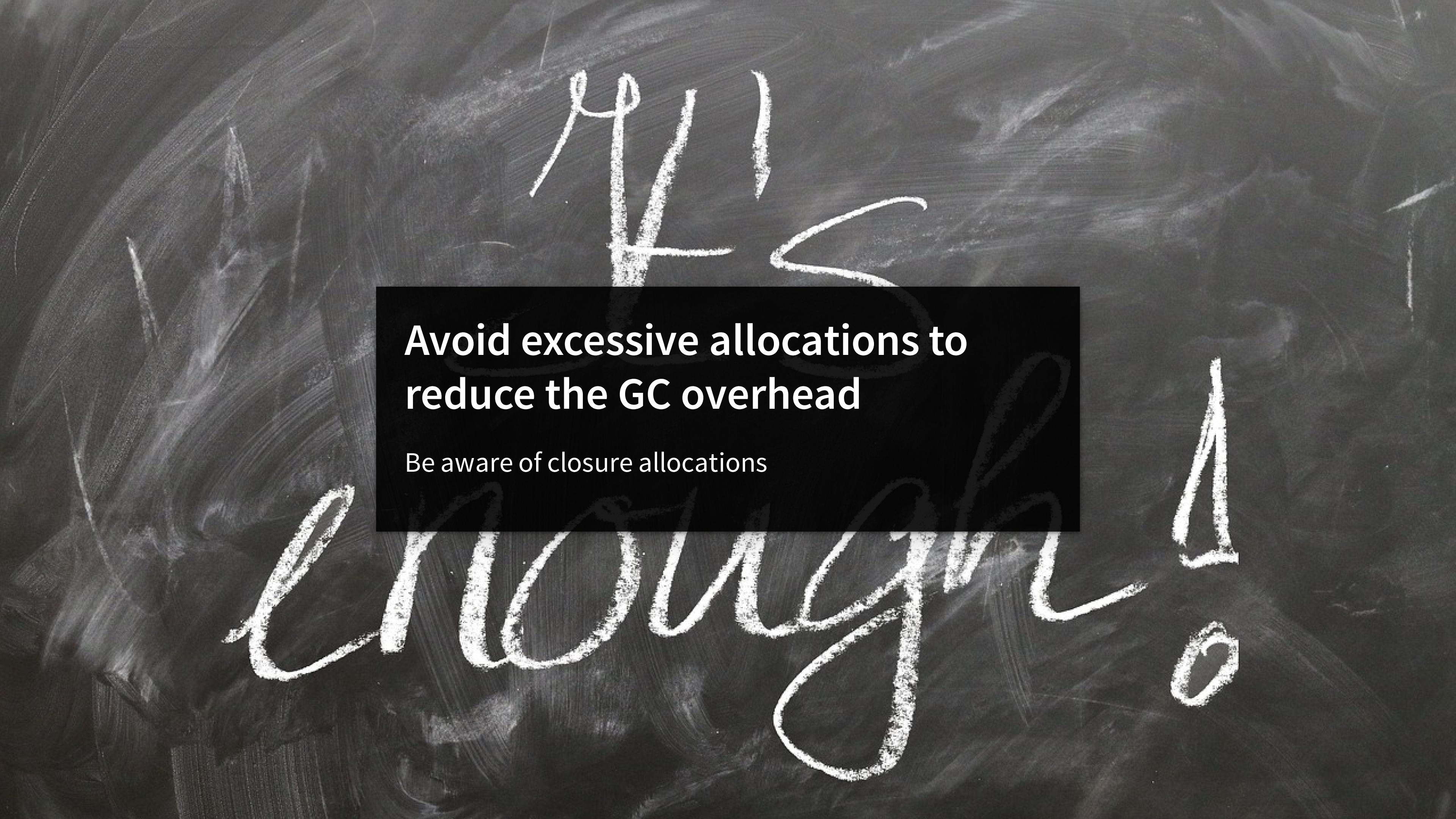
Benchmarking Time!

We can only know the before and after when we measure it.

Method	Size	Collection	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Allocated
AfterV1	0	Array	19.81 ns	29.413 ns	1.612 ns	1.00	0.00	-	-
AfterV2	0	Array	16.35 ns	0.722 ns	0.040 ns	0.83	0.07	-	-
AfterV1	0	Enumerable	20.25 ns	34.142 ns	1.871 ns	1.00	0.00	-	-
AfterV2	0	Enumerable	31.26 ns	76.364 ns	4.186 ns	1.56	0.29	-	-
AfterV1	0	HashSet	53.28 ns	78.903 ns	4.325 ns	1.00	0.00	0.0166	104 B
AfterV2	0	HashSet	29.80 ns	4.916 ns	0.269 ns	0.56	0.04	0.0063	40 B
AfterV1	0	List	37.30 ns	17.950 ns	0.984 ns	1.00	0.00	0.0127	80 B
AfterV2	0	List	26.72 ns	3.334 ns	0.183 ns	0.72	0.02	0.0063	40 B
AfterV1	1	Array	91.51 ns	76.518 ns	4.194 ns	1.00	0.00	0.0229	144 B
AfterV2	1	Array	76.42 ns	18.677 ns	1.024 ns	0.84	0.04	0.0191	120 B
AfterV1	1	Enumerable	112.50 ns	8.332 ns	0.457 ns	1.00	0.00	0.0305	192 B
AfterV2	1	Enumerable	117.57 ns	18.812 ns	1.022 ns	1.05	0.01	0.0241	152 B
AfterV1	1	HashSet	147.98 ns	25.249 ns	1.384 ns	1.00	0.00	0.0446	280 B
AfterV2	1	HashSet	80.78 ns	9.157 ns	0.502 ns	0.55	0.01	0.0204	128 B
AfterV1	1	List	97.80 ns	41.142 ns	2.255 ns	1.00	0.00	0.0267	168 B
AfterV2	1	List	92.70 ns	31.934 ns	1.750 ns	0.95	0.00	0.0204	128 B
AfterV1	4	Array	244.28 ns	92.399 ns	5.065 ns	1.00	0.00	0.0534	336 B
AfterV2	4	Array	350.87 ns	180.579 ns	9.898 ns	1.44	0.01	0.0496	312 B
AfterV1	4	Enumerable	426.69 ns	283.441 ns	15.536 ns	1.00	0.00	0.0610	384 B
AfterV2	4	Enumerable	441.16 ns	108.507 ns	5.948 ns	1.03	0.03	0.0587	368 B
AfterV1	4	HashSet	225.05 ns	12.186 ns	0.668 ns	1.00	0.00	0.0443	280 B
AfterV2	4	HashSet	80.35 ns	28.545 ns	1.565 ns	0.36	0.01	0.0204	128 B
AfterV1	4	List	249.84 ns	89.131 ns	4.886 ns	1.00	0.00	0.0572	360 B
AfterV2	4	List	229.86 ns	98.608 ns	5.405 ns	0.92	0.01	0.0508	320 B

+56%

~23-61%



**Avoid excessive allocations to
reduce the GC overhead**

Be aware of closure allocations


```
1 TransportMessageBatch messageBatch = null;  
2 Task createBatchTask = _retryPolicy.RunOperation(async (timeout)  
3 {  
4     messageBatch =  
5         await CreateMessageBatchInternalAsync(options, timeout);  
6 },  
7 _connectionScope,  
8 cancellationToken);  
9 await createBatchTask;  
10 return messageBatch;
```

Remove closure allocations.

```
1 if (num1 != 0) {
2     this._u003C\u003E8_1 = new AmqpSender.\u003C\u003Ec__DisplayClass16_0();
3     this._u003C\u003E8_1.\u003C\u003E4__this = this.\u003C\u003E4__this;
4     this._u003C\u003E8_1.options = this.options;
5     this._u003C\u003E8_1.messageBatch = (TransportMessageBatch) null;
6
7     configuredTaskAwaiter = amqpSender._retryPolicy.RunOperation(
8         new Func<TimeSpan, Task>((object) this._u003C\u003E8_1,
9             __methodptr(\u003CCreateMessageBatchAsync\u003Eb_0)),
10            (TransportConnectionScope) amqpSender._connectionScope,
11            this.cancellationToken).ConfigureAwait(false).GetAwaiter();
12
13    // rest omitted
14 }
```

Remove closure allocations.


```
1 internal async ValueTask RunOperation<T1>(
2     Func<T1, TimeSpan, CancellationToken, ValueTask> operation,
3     T1 t1,
4     TransportConnectionScope scope,
5     CancellationToken cancellationToken) =>
6     await RunOperation(static async (value, timeout, token) =>
7     {
8         var (t1, operation) = value;
9         await operation(t1, timeout, token);
10        return default(object);
11    },
12    (t1, operation),
13    scope, cancellationToken);
```

Remove closure allocations.

```
1 if (num1 != 0) {
2     configuredTaskAwaiter = t1._retryPolicy
3         .RunOperation<AmqpSender, CreateMessageBatchOptions, TransportMessageBatch>(
4             AmqpSender.\u003C\u003Ec.\u003C\u003E9__16_0 ?? (AmqpSender.\u003C\u003Ec.\u003C\u003E9__16_0 =
5                 new Func<AmqpSender, CreateMessageBatchOptions, TimeSpan, CancellationToken, Task<TransportMessageBatch>>(
6                     (object) AmqpSender.\u003C\u003Ec.\u003C\u003E9,
7                     __methodptr(\u003CCreateMessageBatchAsync\u003Eb__16_0))),
8             t1,
9             this.options,
10            (TransportConnectionScope) t1._connectionScope,
11            this.cancellationToken).ConfigureAwait(false).GetAwaiter();
12     // rest omitted
13 }
```

Remove closure allocations.

Method	Calls	PipelineDepth	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Allocated
V8_PipelineBeforeOptimizations	20000	10	7.083 ms	3.1550 ms	0.1729 ms	1.00	0.00	3054.6875	19,200,023 B
V8_PipelineAfterOptimizations	20000	10	1.588 ms	1.1607 ms	0.0636 ms	0.22	0.01	-	1 B
V8_PipelineBeforeOptimizations	20000	20	10.989 ms	9.0910 ms	0.4983 ms	1.00	0.00	6109.3750	38,400,049 B
V8_PipelineAfterOptimizations	20000	20	2.830 ms	2.4020 ms	0.1338 ms	0.26	0.00	-	2 B
V8_PipelineBeforeOptimizations	20000	40	23.054 ms	11.1449 ms	0.6109 ms	1.00	0.00	12218.7500	76,800,012 B
V8_PipelineAfterOptimizations	20000	40	5.192 ms	4.4372 ms	0.2432 ms	0.23	0.02	-	3 B

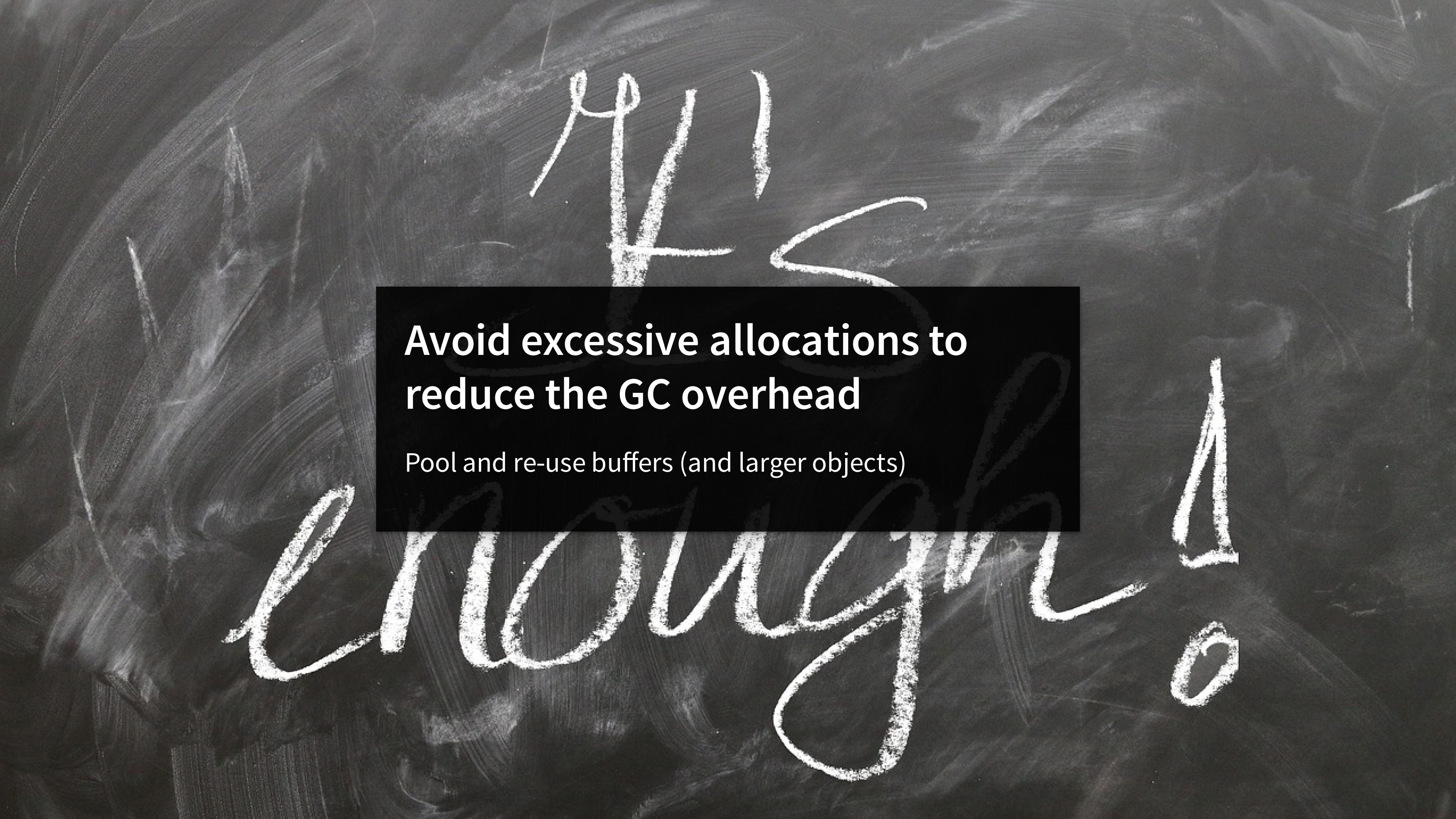
~74-78%
~Gone!

HOW TO DETECT THOSE ALLOCATIONS?

- Use memory profilers and watch out for excessive allocations of *`__DisplayClass`* or various variants of `Action`* and `Func`*
- Use tools like [Heap Allocation Viewer \(Rider\)](#) or [Heap Allocation Analyzer \(Visual Studio\)](#)
- Many built-in .NET types that use delegates have nowadays generic overloads that allow to pass state into the delegate.

```
1 var someState1 = new object();
2 var someOtherState = 42;
3
4 var dictionary = new ConcurrentDictionary<string, string>();
5
6 dictionary.GetOrAdd("SomeKey", static (key, state) => {
7     var (someState, someOtherState) = state;
8
9     return $"{someState}_{someOtherState}";
10 }, (someState1, someOtherState));
```

Remove closure allocations.



**Avoid excessive allocations to
reduce the GC overhead**

Pool and re-use buffers (and larger objects)

```
1 var data = new ArraySegment<byte>(Guid.NewGuid().ToByteArray());
2
3 var guidBuffer = new byte[16];
4 Buffer.BlockCopy(data.Array, data.Offset, guidBuffer, 0, 16);
5 var lockTokenGuid = new Guid(guidBuffer);
```

Pool and re-use buffers.

```
1 byte[] guidBuffer = ArrayPool<byte>.Shared.Rent(16);
2 Buffer.BlockCopy(data.Array, data.Offset, guidBuffer, 0, 16);
3 var lockTokenGuid = new Guid(guidBuffer);
4 ArrayPool<byte>.Shared.Return(guidBuffer);
```

Pool and re-use buffers.

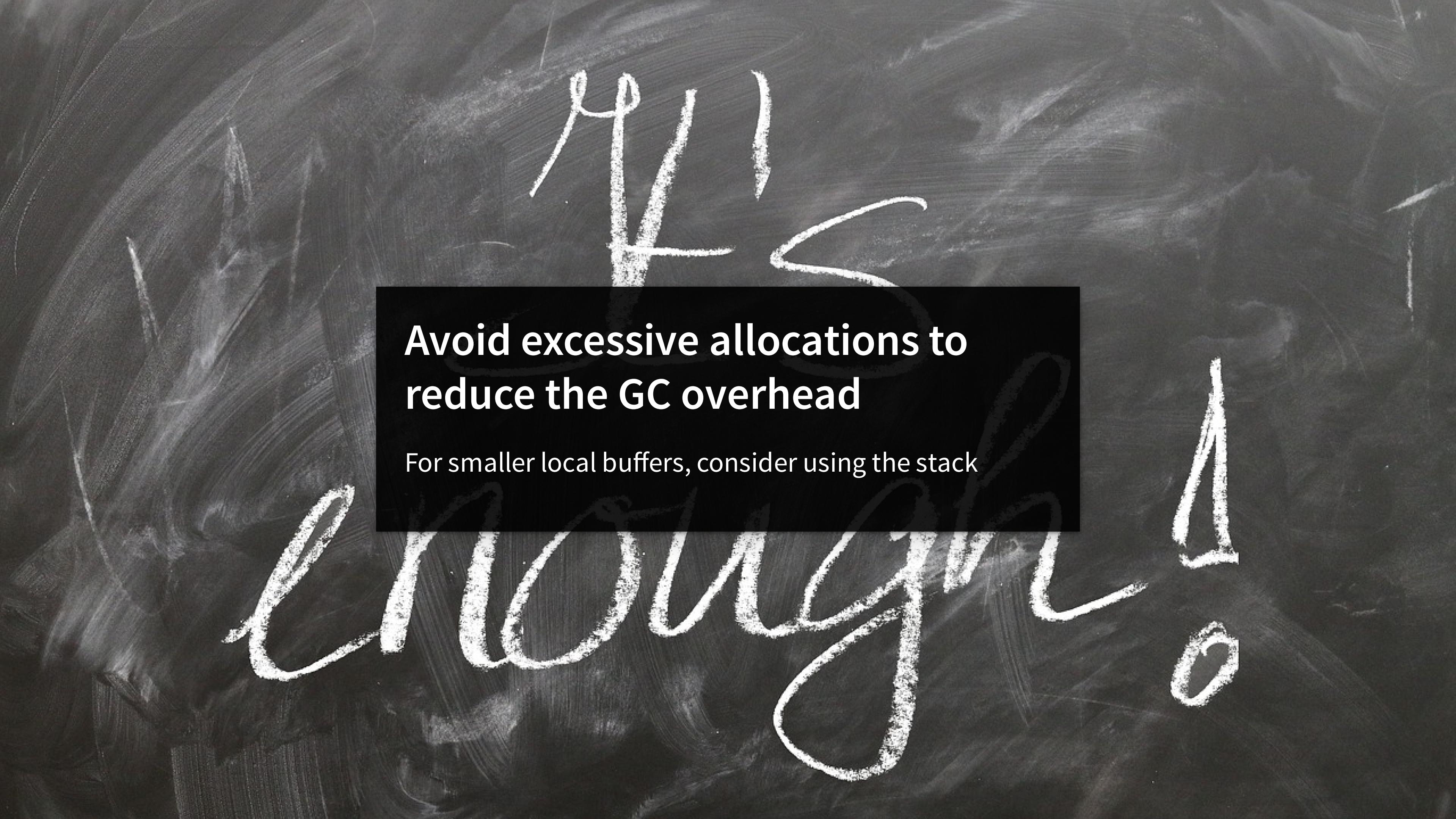


Benchmarking Time!

We can only know the before and after when we measure it.

Method	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Allocated
BufferAndBlockCopy	10.975 ns	0.1860 ns	0.1740 ns	1.00	0.00	0.0064	40 B
BufferPool	24.718 ns	0.3030 ns	0.2555 ns	2.26	0.03	-	-

+226%
~Gone!



**Avoid excessive allocations to
reduce the GC overhead**

For smaller local buffers, consider using the stack

```
1 Span<byte> guidBytes = stackalloc byte[16];  
2 data.AsSpan().CopyTo(guidBytes);  
3 var lockTokenGuid = new Guid(guidBytes);
```

Small local buffers on stack.



Benchmarking Time!

We can only know the before and after when we measure it.

Method	Mean	Error L	StdDev	Ratio	RatioSD	Gen 0	Allocated
BufferAndBlockCopy	10.975 ns	0.1860 ns	0.1740 ns	1.00	0.00	0.0064	40 B
BufferPool	24.718 ns	0.2552 ns	0.2555 ns	2.26	0.03	-	-
StackallocWithGuid	6.078 ns	0.0362 ns	0.0321 ns	0.55	0.01	-	-

A close-up photograph of a person's hand holding a black Sharpie marker, writing the word "Rules" in a bold, black, cursive font on a light-colored wooden surface. The wood has a prominent grain and some darker knots. The hand is positioned in the lower right, with the marker tip pointing towards the bottom left. The background is blurred, showing what appears to be a garden or outdoor setting.

Rules

- Avoid excessive allocations to reduce the GC overhead
 - Think at least twice before using LINQ or unnecessary enumeration on the hot path
 - Be aware of closure allocations
 - Pool and re-use buffers
 - For smaller local buffers, consider using the stack
 - Be aware of parameter overloads
 - Where possible and feasible use value types but pay attention to unnecessary boxing



Avoid unnecessary copying of
memory

Avoid unnecessary copying of memory

- Look for Stream and Byte-Array usages that are copied or manipulated without using Span or Memory
- Replace existing data manipulation methods with newer Span or Memory based variants

```
1 private static short GenerateHashCode(string partitionKey) {
2     if (partitionKey == null) {
3         return 0;
4     }
5
6     var encoding = Encoding.UTF8;
7     ComputeHash(encoding.GetBytes(partitionKey), 0, 0, out uint hash1, out uint hash2);
8     return (short)(hash1 ^ hash2);
9 }
10
11 private static void ComputeHash(byte[] data, uint seed1, uint seed2,
12     out uint hash1, out uint hash2) {
13
14     uint a, b, c;
15
16     a = b = c = (uint)(0xdeadbeef + data.Length + seed1);
17     c += seed2;
18
19     int index = 0, size = data.Length;
20     while (size > 12) {
21         a += BitConverter.ToInt32(data, index);
22         b += BitConverter.ToInt32(data, index + 4);
23         c += BitConverter.ToInt32(data, index + 8);
24
25     // rest omitted
26 }
```

Avoid unnecessary copying of memory.



Benchmarking Time!

We can only know the before and after when we measure it.

Method	Size	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Allocated
Before	8	30.46 ns	10.493 ns	0.575 ns	1.00	0.00	0.0010	32 B
After	8	18.47 ns	2.863 ns	0.157 ns	0.61	0.02	-	-
Before	12	31.88 ns	5.089 ns	0.279 ns	1.00	0.00	0.0012	40 B
After	12	18.71 ns	2.281 ns	0.125 ns	0.59	0.01	-	-
Before	24	38.95 ns	3.828 ns	0.210 ns	1.00	0.00	0.0014	48 B
After	24	24.19 ns	0.476 ns	0.026 ns	0.62	0.00	-	-
Before	32	45.05 ns	6.978 ns	0.383 ns	1.00	0.00	0.0017	56 B
After	32	27.53 ns	1.200 ns	0.61	0.61	0.00	-	-
Before	64	74.93 ns	7.068 ns	0.387 ns	1.00	0.00	0.0026	38 B
After	64	39.59 ns	1.935 ns	0.106 ns	0.53	0.00	-	-
Before	128	123.84 ns	12.736 ns	0.698 ns	1.00	0.00	0.0045	152 B
After	128	78.06 ns	2.420 ns	0.133 ns	0.63	0.00	-	-
Before	255	199.68 ns	11.492 ns	0.630 ns	1.00	0.00	0.0083	280 B
After	255	126.24 ns	1.639 ns	0.090 ns	0.63	0.00	-	-
Before	257	206.22 ns	8.201 ns	0.450 ns	1.00	0.00	0.0086	288 B
After	257	128.98 ns	2.596 ns	0.142 ns	0.63	0.00	-	-
Before	512	380.75 ns	74.100 ns	4.062 ns	1.00	0.00	0.0157	536 B
After	512	212.34 ns	14.202 ns	0.778 ns	0.56	0.01	-	-



Rules



- Look for Stream and Byte-Array usages that are copied or manipulated without using Span or Memory
- Replace existing data manipulation methods with newer Span or Memory based variants
- Watch out for immutable/readonly data that is copied

- Avoid excessive allocations to reduce the GC overhead
 - Be aware of closure allocations
 - Be aware of parameter overloads
 - Where possible and feasible use value types but pay attention to unnecessary boxing
 - Think at least twice before using LINQ or unnecessary enumeration on the hot path
 - Use `Array.Empty<T>` to represent empty arrays
 - Use `Enumerable.Empty<T>` to represent empty enumerables
 - Prevent collections from growing
 - Use concrete collection types
 - Leverage pattern matching or `Enumerable.TryGetNonEnumeratedCount`
 - Wait with instantiating collections until really needed
 - Pool and re-use buffers
 - For smaller local buffers, consider using the stack
- Avoid unnecessary copying of memory
 - Watch out for immutable/readonly data that is copied
 - Look for Stream and Byte-Array usages that are copied or manipulated without using Span or Memory
 - Replace existing data manipulation methods with newer Span or Memory based variants

In case you are up for a challenge

The original `ComputeHash` method had a .

Did you spot it?

```
static void ComputeHash(byte[] data, uint seed1, uint seed2,
    out uint hash1, out uint hash2) {

    uint a, b, c;

    a = b = c = (uint)(0xdeadbeef + data.Length + seed1);
    c += seed2;

    int index = 0, size = data.Length;
    while (size > 12) {
        a += BitConverter.ToInt32(data, index);
        b += BitConverter.ToInt32(data, index + 4);
        c += BitConverter.ToInt32(data, index + 8);

        // rest omitted
    }
}
```

Strategy Innovation

AT SCALE IMPLEMENTATION DETAILS MATTER

Tweak expensive I/O operations first.

Pay close attention to the context of the code.

Apply the principles where they matter.

Everywhere else, favor readability.

github.com/danielmarbach/PerformanceTricksAzureSDK

Happy coding!

✉ [@danielmarbach](https://twitter.com/danielmarbach) ✉ daniel.marbach@particular.net

