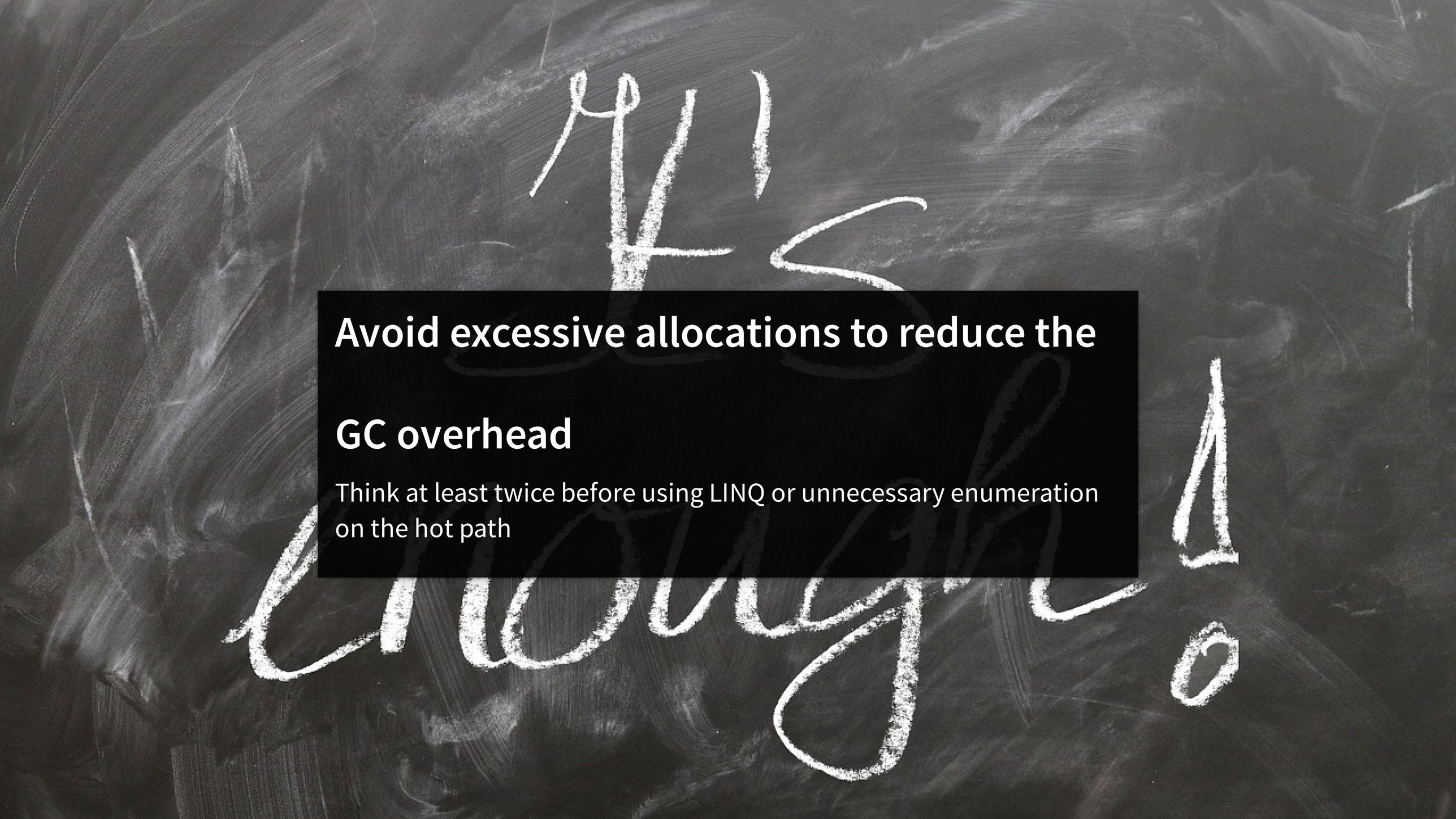
Performance tricks I learned from contributing to open source .NET packages

danielmarbach | daniel.marbach@particular.net

ESOTERIC

Being called out for premature optimizations.



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 class AmqpReceiver {
2
3    // ...
4    // Compiler generated chunk we are not really interested in right now
5
6    private Task CompleteInternalAsync(IEnumerable<string> lockTokens)
7    {
8      Enumerable.Any(Enumerable.ToArray(Enumerable.Select(lockTokens, <>c.<>9__2_0 ??
9        (<>c.<>9__2_0 = new Func<string, Guid>(<>c.<>9.<CompleteInternalAsync>b__2_0)))),
10         new Func<Guid, bool>(<CompleteInternalAsync>b__2_1));
11     return Task.CompletedTask;
12   }
13
14   [CompilerGenerated]
15   private bool <CompleteInternalAsync>b__2_1(Guid lockToken)
16   {
17     return Enumerable.Contains(_lockedMessages, lockToken);
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[] lockTokenGuids = lockTokens.Select(token => new Guid(token)).ToArray();
7    foreach (var tokenGuid in lockTokenGuids)
8    {
9      if (_requestResponseLockedMessages.Contains(tokenGuid))
10     {
11       return Task.CompletedTask;
12     }
13   }
14   return Task.CompletedTask;
15 }
```

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))));
9
10   int num = 0;
11   while (num < array.Length)
12   {
13     Guid item = array[num];
14     if (_requestResponseLockedMessages.Contains(item))
15     {
16       return Task.CompletedTask;
17     }
18     num++;
19   }
20   return Task.CompletedTask;
21 }
```

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	7[illegible] ns	[illegible] ns	1.00	0.00	0.0381	240 B
AfterV1	1	Array	93.27 ns	25.817 ns	1.505 ns	0.73	0.02	0.0229	144 B
Before	1	Enumerable	147.40 ns	14[illegible] 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

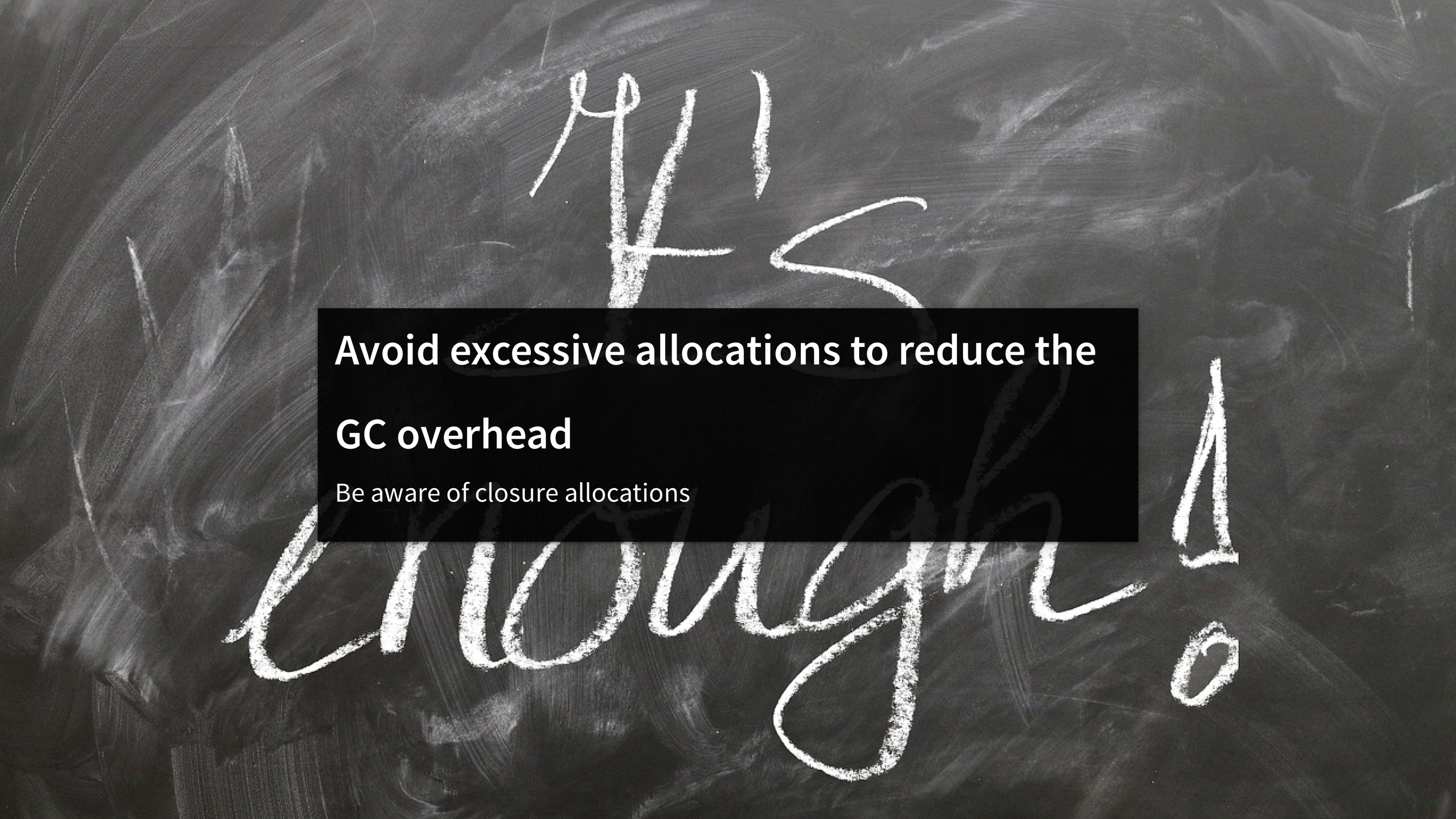
LINQ TO COLLECTION-BASED OPERATIONS

- 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

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.[illegible] ns	[illegible] ns	1.00	0.00	0.0305	192 B
AfterV2	1	Enumerable	117.57 ns	18.823 ns	1.032 ns	1.05	0.01	0.0241	152 B
AfterV1	1	HashSet	147.98 ns	[illegible] ns	[illegible] 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 var someState = new object();
2 var someOtherState = 42;
3
4 var dictionary = new ConcurrentDictionary<string, string>();
5
6 dictionary.GetOrAdd("SomeKey", (key) =>
7 {
8   return $"{someState}_{someOtherState}";
9 });
```

Remove closure allocations.

```
1 <>c__DisplayClass0_0 <>c__DisplayClass0_ = new <>c__DisplayClass0_0();
2 <>c__DisplayClass0_.someState = new object();
3 <>c__DisplayClass0_.someOtherState = 42;
4
5 concurrentDictionary.GetOrAdd("SomeKey",
6   new Func<string, string>(<>c__DisplayClass0_.<<Main>$>b__0));
```

Remove closure allocations.

```
// same as before

dictionary.GetOrAdd("SomeKey", static (key, state) =>
{
  var (someState, someOtherState) = state;

  return $"{someState}_{someOtherState}";
}, (someState, someOtherState));
```

Remove closure allocations.

```
object item = new object();
int item2 = 42;

concurrentDictionary.GetOrAdd("SomeKey",
  <>c.<>9__0_0 ?? (<>c.<>9__0_0 =
    new Func<string, ValueTuple<object, int>, string>(<>c.<>9.<<Main>$>b__0_0)),
      new ValueTuple<object, int>(item, item2));
```

Remove closure allocations.

Benchmarking Time!

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

Method	Mean	Error	StdDev	Ratio	Gen 0	Allocated
GetOrAddWithClosure	45.84 ns	0.680 ns	0.347 ns	1.00	0.0102	64 B
GetOrAddWithoutClosure	13.37 ns	0.619 ns	0.549 ns	0.29	-	-

~71%

~Gone!

HOW TO DETECT THOSE ALLOCATIONS?

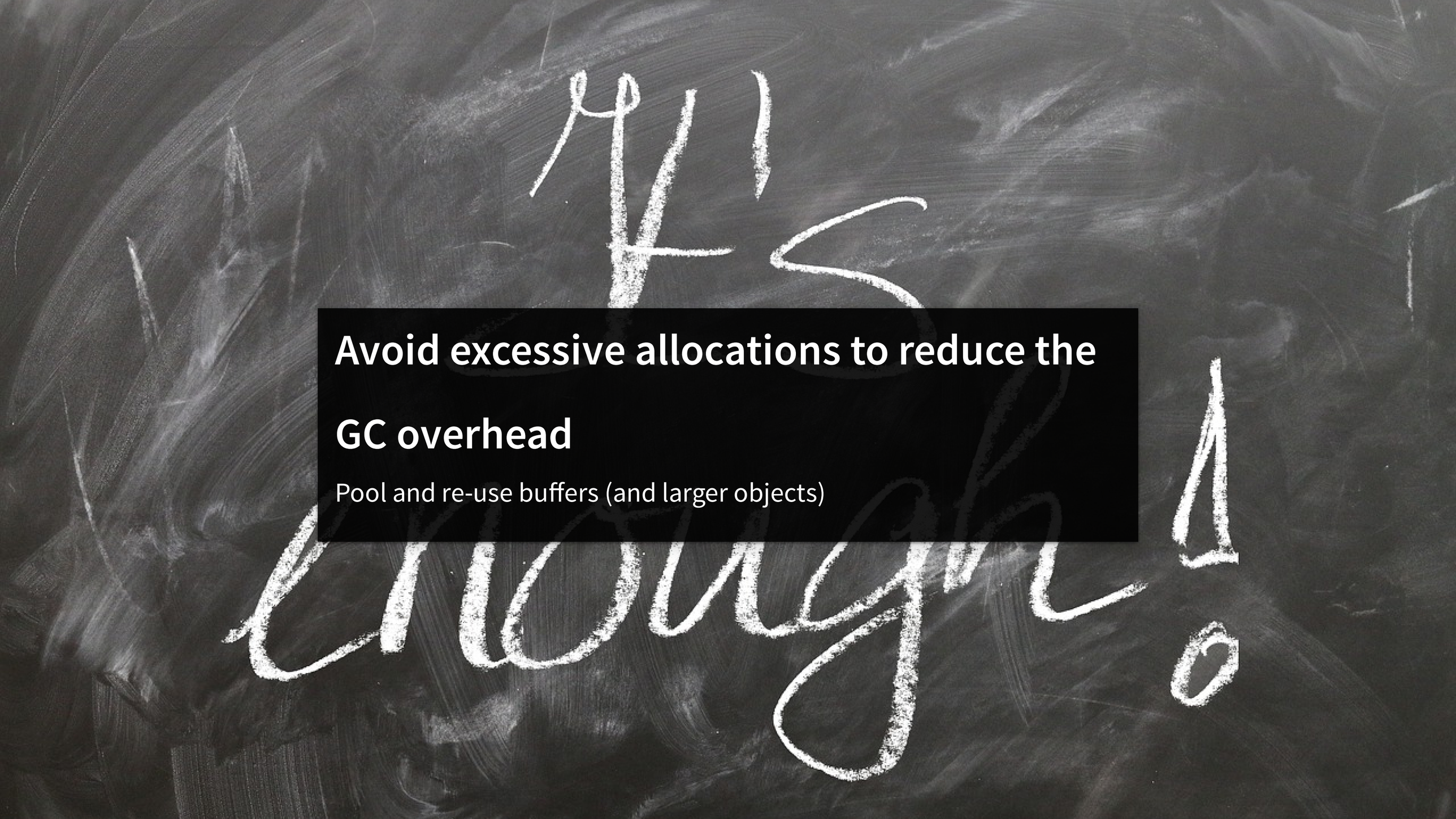
- Use memory profilers and watch out for excessive allocations of `*__DisplayClass*` or various variants of `Action*` and `Func*`
- Use tools like Heap Allocation Viewer (Rider) or Heap Allocation Analyzer (Visual Studio)
- Many built-in .NET types that use delegates have nowadays generic overloads that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

~74-78%

~Gone!

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	2.4414 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

particular.net/blog/pipeline-and-closure-allocations



Avoid excessive allocations to reduce the GC overhead

Pool and re-use buffers (and larger objects)

```
var data = new ArraySegment<byte>(Guid.NewGuid().ToByteArray());

var guidBuffer = new byte[16];
Buffer.BlockCopy(data.Array, data.Offset, guidBuffer, 0, 16);
var lockTokenGuid = new Guid(guidBuffer);
```

Pool and re-use buffers.

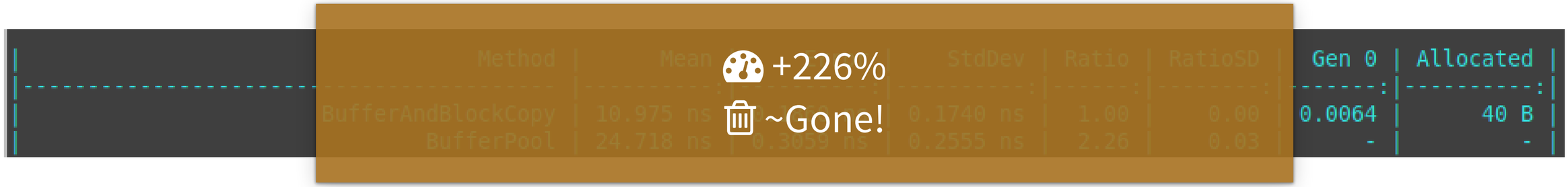
```
1 byte[] guidBuffer =  ArrayPool<byte>.Shared.Rent(16);
2 Buffer.BlockCopy(data.Array, data.Offset, guidBuffer, 0, 16);

3 var lockTokenGuid = new Guid(guidBuffer);
4 ArrayPool<byte>.Shared.Return(guidBuffer);
```

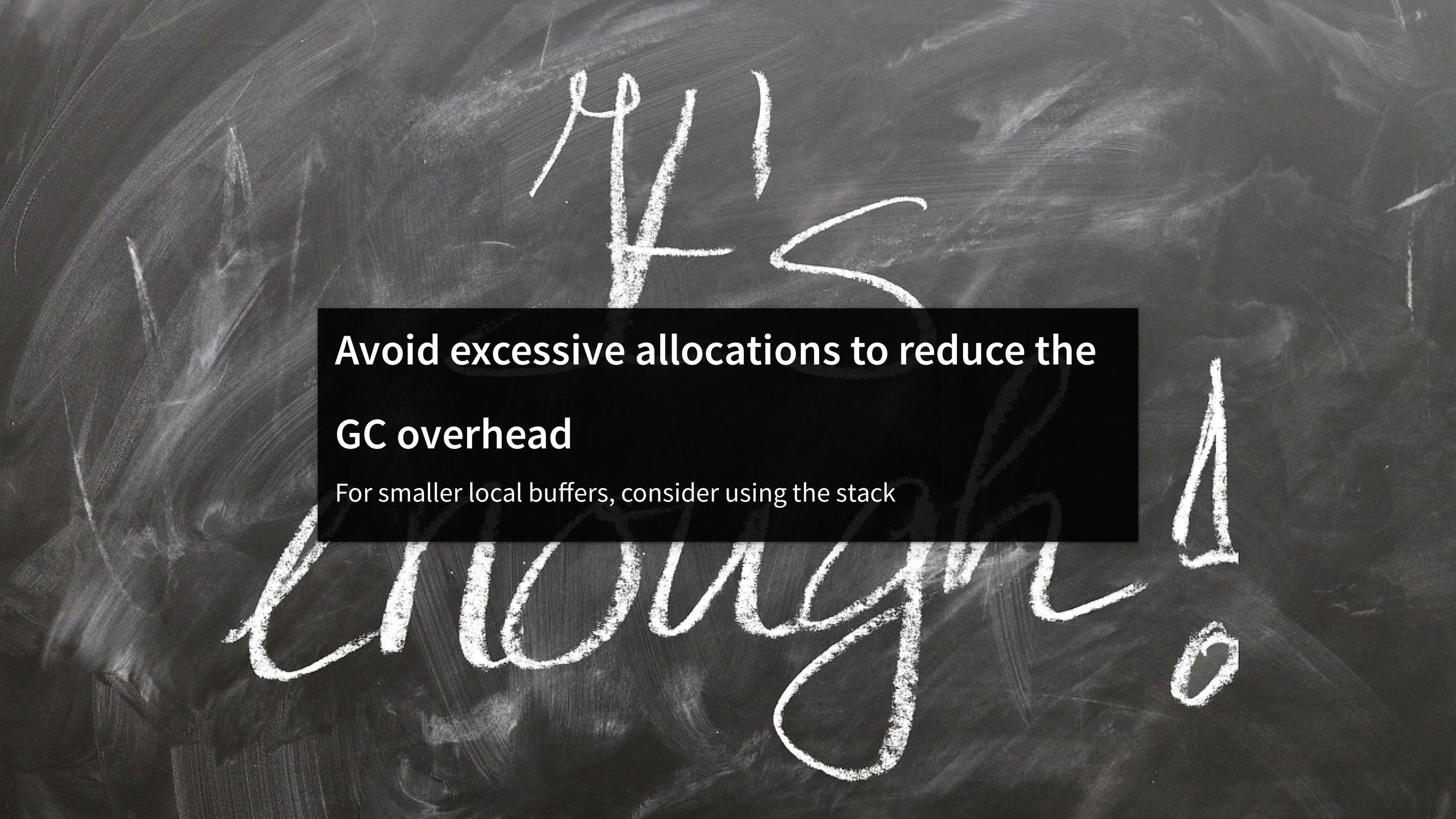
Pool and re-use buffers.

Benchmarking Time!

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



Method	Mean	StdDev	Ratio	RatioSD	Gen 0	Allocated
BufferAndBlockCopy	10.975 ns	0.1740 ns	1.00	0.00	0.0064	40 B
BufferPool	24.718 ns	0.3059 ns	0.2555 ns	2.26	0.03	-	-


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	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	0.2500 ns	0.2555 ns	2.26	0.03	-	-
StackallocWithGuid	6.078 ns	0.0362 ns	0.0321 ns	0.55	0.01	-	-

~45%

~Gone!


Rules

- Avoid excessive allocations to reduce the GC overhead
  - Be aware of closure allocations
  - 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

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 daniel.marbach@particular.net