

<http://tech.pro/tutorial/1433/performance-benchmark-mistakes-part-four>


AUG

JAN

MAY

14

2013 2014 2015

About this capture

12 captures  
19 Aug 2013 - 15 Jun 2016

CONTENT

NETWORK

MORE

Create a Profile or Login

Search Tech.Pro

How Tech.Pro Works



The nerd table is now the cool table.



**Eric Lippert**  
posted 6 years ago



BEGINNER

Performance

Likes

5

Views

5.8k

## ABOUT THE AUTHOR



**Eric Lippert**  
Follow

Eric Lippert designs C# analyzers at Coverity and formerly designed languages at Microsoft. Learn more at <http://ericlippert.com>.

## WANT TO WRITE WITH US?

Anyone can write at Tech.Pro. Writing at Tech.Pro can help you get recognized, spread knowledge, and inspire others.

## INVITE A FRIEND

Tech.Pro is a communications platform and will become more useful to you as your colleagues join the site. Go ahead and invite a few today and start reaping the benefits of membership.



Dear Tech Professional,

Tech.pro has been temporarily deactivated. We will bring Tech.pro back online after our team has grown in sufficient size to support the community.

Sincerely,  
Tech.pro Team



Last time in this series we talked about how forgetting to "charge" the expense of the jitter to your analysis can lead to inaccurate results: since the first run of a method incurs the cost of the jitter, you might want to leave that out of your average. Or you might want to leave it in! It depends on your scenario. But regardless, it is wise to be aware that there is a cost there, and that it needs to be taken into account when reasoning about measurements.

The same is true of garbage collection, which is the subject of today's episode of this tutorial. So without

further ado:

### Mistake #8: Forget to take garbage collection effects into account.

Before we get into how to mitigate the situation, a brief explanation of how the .NET garbage collector works is in order. The explanation which follows is somewhat simplified but gets the necessary ideas across.

Imagine a large block of memory, far larger than the size of any typical C# object. Roughly halfway through that block of memory is a memory location called the "high water mark". Everything below the high water mark is the storage for the fields of objects and the elements of arrays. Above the high water mark, the remainder of the memory block is empty, just thousands and thousands of zero bits. This is the managed heap.

When you allocate a new array, or a new object of reference type, or box an instance of a value type, the storage for the array elements, object fields or boxed value has to go somewhere. The high water mark is moved up by the required number of bytes and the storage is passed back to the user to be initialized with the array elements, or to be filled in by the constructor, or to have the value copied in.

This can't go on forever of course. What happens when the high water mark reaches the top of the block?

This triggers a collection. All the threads in the program are temporarily frozen, because they might be using memory that the garbage collector is about to mess with. (In some versions of the CLR it is possible to continue running code while the GC is running, but we will not discuss this advanced scenario.) The garbage collector, which runs on a thread of its own, is activated. It sets a bit on every object in the memory block that says "this object is dead". It knows what objects are "roots"; that is, what objects are known definitely to be in use by the program at this time. It clears the bit on those objects, and then recursively clears the bit on every object referenced by those objects, and so on. Before long, every object that

<http://tech.pro/tutorial/1433/performance-benchmark-mistakes-part-four>

Go

AUG JAN MAY

14

2013 2014 2015



About this capture

12 captures

19 Aug 2013 - 15 Jun 2016

Subscribe

Next, any of the dead objects that have finalizers that need to run are marked as being alive and moved onto a special queue. The finalizer queue is a root of the garbage collector, so those objects will stay alive until their finalizers run (on yet another thread) at which time they will be removed from the queue. At that time they will likely be truly dead, and the garbage collector will collect them.

Finally, the dead objects are now "holes" that can be filled in by living objects. The garbage collector moves the living objects near the high water mark into the holes, and thereby lowers the high water mark.

So far what I've described is a *single generation mark-and-sweep collector*. The "desktop" .NET garbage collector is actually more sophisticated than that. It actually has three large blocks of memory, called the generation 0, generation 1 and generation 2 heaps. When generation 0 is collected, the surviving objects are not copied back into the holes; rather, they are copied up to generation 1, and the generation 0 block becomes empty. When the generation 1 heap is collected, again, the survivors are copied up to generation 2.

The idea here is that we are identifying objects that are short lived: everything that was dead in generation 0 was very short lived. Objects collected in generation 1 were longer-lived, and objects that make it all the way to generation 2 are extremely long-lived.

Because garbage collection requires tracing the set of living objects, the cost of garbage collection is gated on the number of living objects. By using a generational approach, inexpensive generation 0 collections happen frequently; these are the most likely to produce garbage, and the number of living generation 0 objects is likely to be small. By contrast, long-lived objects that make it to generation 2 are the least likely to be released on any given collection. If there are a lot of them then collection will be expensive and might not actually free up much space. The runtime therefore wants to run a generation 2 collection as infrequently as it can get away with.

So, summing up, the takeaways here are:

- Garbage collection stops the world while the collection happens
- Generation 0 collections are frequent but inexpensive; generation 2 collections are infrequent and likely to be expensive; generation 1 collections are in the middle.
- Objects with finalizers automatically survive; the finalizers run on a different thread, after the collection happens.

How then does this impact performance benchmarks? Let's think of a few different scenarios.

**Scenario one:** Your benchmark produces only short-lived garbage, and the same number of collections are triggered every time you run the benchmark. This is the best situation you can be in, because every time through the benchmark you are measuring the actual cost that would be observed by a user. Unfortunately, things seldom work out so tidily.

**Scenario two:** Your benchmark produces only short-lived garbage, and moreover, it produces enough short-lived garbage to trigger a collection, let's say every ten times the benchmark is executed. If you are running the benchmark fewer than ten times, essentially this means that you are never measuring the cost of a collection because you're never triggering one. If you are running the benchmark a hundred times then roughly every tenth run will be more expensive than the previous nine.

This has both positive and negative effects. On the positive side, you can now compute both the average cost and the maximum typical cost, which is useful data. On the negative side, it can be confusing to see variations like this in your measurements if it is not clear what is going on.

**Scenario three:** Now consider scenarios one and two again, but suppose this time that the benchmark produces long-lived garbage that is infrequently collected, and suppose some of

could be using. This is particularly of concern if there are more active threads in your program than hardware processors to service them.

**Scenario four:** Your benchmark is actually testing two different functions, and comparing their costs; benchmarks are often used for comparative testing. Suppose the garbage produced by the first method is collected during the execution of the second method; now we're making an inaccurate comparison of the two alternatives because one is being charged a cost that was incurred by the other.

**Scenario five:** Your benchmark mechanisms themselves are producing garbage, causing collections to happen more often than normal during the code that you are benchmarking. Again, you might not be measuring the true cost of a piece of code, because this time the costs of the benchmarking environment are being charged to the code being tested.

In short, failing to take GC costs into account can cause you to fail to measure the true cost of an operation, or cause you to charge that cost to the wrong code.

So how do you mitigate these problems?

What I usually do when benchmarking, particularly if it is comparative benchmarking as described in scenario four, is to force the garbage collector to do a full collection of all three generations before and after every test. This ensures that every test is on the level playing field.

You might consider measuring the costs of each of those collections as well, to try to get a handle on just how much time you're spending on every full collection. Remember, full collections are going to be more expensive than the partial collections that are normally performed, so this will give you an upper bound on the cost, not the true cost.

To force the garbage collector to do a full collection you should actually do two steps:

```
System.GC.Collect();  
System.GC.WaitForPendingFinalizers();
```

The first line causes all three generations to be collected; all threads are stopped and the GC reclaims everything that it can. But that might have caused objects to be placed on the finalizer queue, which runs on its own thread. In order to make sure we are measuring the cost of finalization, we stop the world again and wait for the finalizers to all run.

If you've been paying close attention so far, something should be nagging you at this point. Remember that I said that objects on the finalizer queue are alive until they are finalized, and therefore they live until a collection happens after their finalization. By doing these two steps we are not actually guaranteeing that every dead object has been cleaned up; an object that was dead that needed finalization before the call to `GC.Collect()` will be alive after the call, and then dead after the call to `GC.WaitForPendingFinalizers()`, but not cleaned up yet because the garbage collector just did a collection; it's not going to run again for a while.

Usually the number of objects that survive to the next generation because they need finalization is extremely small, so this situation usually does not produce a measurable impact. However if your benchmarked method produces a huge amount of garbage that needs finalization, you might consider doing a *second* `GC.Collect()` afterwards, to ensure that objects that survived only to be finalized are cleaned up. (And if you are producing a huge amount of garbage that needs finalization, there's probably something that can be improved in your program. But that's a subject for another day.)

Next time, we'll finish up this series with a look at ways to make sure that your measurements are repeatable and realistic.



Dear Tech Professional,

Tech.pro has been temporarily deactivated. We will bring Tech.pro back online after our team has grown in sufficient size to support the community.

Sincerely,  
Tech.pro Team

Login

Email Address

Password

Login

Register

Email Address

First Name

Last Name

Password

☐ I agree to Tech.Pro's Terms

Register

Comments (2)



**Doron Grinzaig** posted 6 years ago

I might be missing something obvious here but... You said:

Usually the number of objects that survive to the next generation because they need finalization is extremely small, so this situation usually does not produce a measurable impact.

Then why do you even need to wait for those objects with: `WaitForPendingFinalizers` ?

Thanks.

↗ Reply



**gopal krishan** posted 6 years ago

But reality is not that simple. Garbage collection heap is self tuning. After first garbage collection, the high water mark may have been changed. So even after doing GC.Collect, second function might be running with a different playing field than the first function.

↗ Reply




AUGJANMAY



◀14▶

201320142015

[12 captures](#)

19 Aug 2013 - 15 Jun 2016





About this capture