

<http://tech.pro/tutorial/1317/c-performance-benchmark-mistakes-part-three>

Go

SEP DEC MAY

07

2012 2013 2014



About this capture

13 captures

13 Jul 2013 - 3 Mar 2019

CONTENT

NETWORK

MORE

Create a Profile or Login

Search Tech.Pro

How Tech.Pro Works



The nerd table is now the cool table.

learn more

Eric Lippert  
posted 6 years ago

BEGINNER

C# Performance Benchmark

Likes 8

Views 6k

# C# Performance Benchmark Mistakes, Part Three



As I discussed in part one of this series, a C# program actually goes through two compilers before the code runs. First the C# compiler converts the code into "Common Intermediate Language", or "IL", which is written into the *assembly* -- the .EXE or .DLL file of the program. Then at runtime, immediately before a method is called for the first time, the aptly-named "just in time" compiler -- the jitter, for short -- transforms the IL into actual executable machine code. Because computer programmers love to verb words, this process is called "jitting a method".

The jitter was built to be fast, but it's not infinitely fast. You should expect that the first time you run a method it will be considerably slower; before it can run for the first time all its IL has to be translated into machine code. And of course, if the methods that *it* calls in turn are all being called for the first time then they've got to be jitted as well.

But wait, it gets worse. Suppose you are calling a method for the first time, and it in turn creates an object from a library. If that library is being used for the first time in this process, not only does the code for the constructor have to be jitted, but before it can be jitted the library has to be found on disk, mapped into virtual memory, analyzed for security problems, and any code in the static constructors of the newly-created object has to be jitted and executed.

In short, calling code for the first time can be massively more expensive than calling it for the second time. So this brings us to...

## Mistake #6: Treat the first run as nothing special when measuring average performance.

In order to get a good result out of a benchmark test in a world with potentially expensive startup costs due to jitting code, loading libraries and calling static constructors, you've got to apply some careful thought about what you're actually measuring.

If, for example, you are benchmarking for the specific purpose of *analyzing startup costs* then you're going to want to make sure that you measure *only* the first run. If on the other hand you are benchmarking part of a service that is going to be running millions of times over many days and you wish to know the average time that will be taken in a typical usage then the high cost of the first run is irrelevant and therefore *shouldn't be part of the average*. Whether you include the first run in your timings or not is up to you; my point is, you need to be cognizant of the fact that the first run has potentially very different costs than the second.

Let's illustrate the enormous effect that the jitter can have on the first run of a program by modifying our example from last time. Let's suppose we've written our own implementation

### 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.

Write &amp; Win

### 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.

Invite a Friend



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

TECH.PRO DAILY EMAIL

<http://tech.pro/tutorial/1317/c-performance-benchmark-mistakes-part-three>

Go

SEP DEC MAY

07

2012 2013 2014



About this capture

13 captures

13 Jul 2013 - 3 Mar 2019

Subscribe

```

{
    // Find an item greater than the pivot
    while (items[left] < pivot)
        left += 1;
    // Find an item less than the pivot
    while (pivot < items[right])
        right -= 1;

    // Swap them
    if (left <= right)
    {
        Swap(ref items[left], ref items[right]);
        left += 1;
        right -= 1;
    }
}

// Everything less than the pivot is now left of the pivot.
// Sort this portion.
if (leftBound < right)
    Quicksort(items, leftBound, right);
// Same for the right.
if (left < rightBound)
    Quicksort(items, left, rightBound);
}
private static void Swap(ref int x, ref int y)
{
    int t = x;
    x = y;
    y = t;
}

```

That's a pretty standard implementation of quicksort in C#. Now let's make some modifications to our benchmark test of last time. Here we'll make a random list, copy it to an array twice so that we know that both runs will be operating on exactly the same data, and see if we can deduce the cost of jitting those methods the first time they're called:

```

class P
{
    static void Main()
    {
        var original = new List<int>();
        const int size = 1000;
        var random = new Random();
        for(int i = 0; i < size; ++i)
            original.Add(random.Next());

        var arr1 = original.ToArray();
        var arr2 = original.ToArray();

        var stopwatch = new System.Diagnostics.Stopwatch();
        stopwatch.Start();
        arr1.Quicksort();
        stopwatch.Stop();
        Console.WriteLine(stopwatch.Elapsed);
        stopwatch.Reset();
        stopwatch.Start();
        arr2.Quicksort();
        stopwatch.Stop();
        Console.WriteLine(stopwatch.Elapsed);
    }
}

```

When I run this on my laptop (again, remembering to compile into release mode, and running without the debugger attached) a typical result is 3500 microseconds for the first run and 700 microseconds for the second run; in other words, the first run here took roughly *five times longer* than the second run on average. It must have taken the jitter about 2.8 milliseconds on average to find the IL and translate it into efficient machine code.

Of course, that factor is relative, and I chose the array size somewhat arbitrarily. If we were sorting an array with a million elements then the ~3 millisecond jit cost would be a barely-

<http://tech.pro/tutorial/1317/c-performance-benchmark-mistakes-part-three>

Go

SEP DEC MAY

07

2012 2013 2014



About this capture

[13 captures](#)

13 Jul 2013 - 3 Mar 2019

Moreover, it's important to note that different jitters give different results on different machines and in different versions of the .NET framework. The time taken to jit can vary greatly, as can the amount of optimization generated in the machine code. The jit compilers on the Windows 32 bit desktop, Windows 64 bit desktop, Silverlight running on a Mac, and the "compact" jitter that runs when you have a C# program in XNA on XBOX 360 all have potentially different performance characteristics. That's...

#### Mistake #7: Assuming that runtime characteristics in one environment tell you what behavior will be in a different environment.

Run your benchmarks in the actual environment that the code will be running in; use machines that have the same hardware and software that will typically be used by the customers who ultimately will run the code.

Next time in this series we'll take a look at how the garbage collector can affect performance benchmarks.

[Like this post](#)

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

### Register

☐ I agree to [Tech.Pro's Terms](#)

### Comments (13)



**Cosmin Ivan** posted 6 years ago

What if there are other processes running "randomly" in the background? Won't they influence the time measured?



**Eric Lippert** 6 years ago

Yes! I'll likely discuss that in a future episode.



**Ilya Ivanov** posted 6 years ago

Very nice article, as always. Just a tiny note to make code cleaner and shorter: instead of creating new Stopwatch instance and using Start method to start it, you can use Stopwatch.StartNew() static method, which will do all this for you in one line.

<http://tech.pro/tutorial/1317/c-performance-benchmark-mistakes-part-three>

Go

SEP DEC MAY

07

2012 2013 2014



About this capture

[13 captures](#)

13 Jul 2013 - 3 Mar 2019

**Vitor Canova** posted 6 years ago

Ok, got that. Just wondered if it had another magic answer. You did this to me when you said StringBuilder is created in a loop for JS coders. You spoiled me ;)

[➔ Reply](#)**Eric Lippert** posted 6 years ago

Interesting; I had not heard about that. Thanks!

[➔ Reply](#)**Allon Guralnek** posted 6 years ago

Great article as always, Eric, keeping them coming. I believe the 2 GB limitation for CLR arrays was lifted in .NET 4.5, was it not? See <http://msdn.microsoft.com/en-us/library/hh285054.aspx>

[➔ Reply](#)**Eric Lippert** posted 6 years ago

I like to write code that doesn't lay traps for the unwary. Let's suppose we said:

```
stopwatch.Start(); arr1.Quicksort(); Console.WriteLine(stopwatch.Elapsed);
```

as you suggest. Now I want to change the code to:

```
stopwatch.Start(); arr1.Quicksort(); Console.WriteLine("Sorting array {0} took {1}",  
original.MakeString(), stopwatch.Elapsed);
```

And hey, now I'm not measuring what I thought I was measuring anymore.

You don't get points for making the code as short as possible. Make the code as *maintainable* as possible; that saves you time.

[➔ Reply](#)**Vitor Canova** posted 6 years ago

You really need to call stopwatch.Stop() before consume the value inside stopwatch.Elapsed?

To me it looks like pointless since you allied stopwatch.Reset() just after.

[➔ Reply](#)**Erik Forbes** posted 6 years ago

"Because computer programmers love to verb words" - I see what you did there.

[➔ Reply](#)**Eric Lippert** posted 6 years ago

Good point, though I note that in practice this bug never happens. For leftBound + rightBound to be greater than  $2^{31}$ , leftBound and rightBound must be on the order of  $2^{30}$ . That implies that there's an integer array with  $2^{30}$  elements, which is impossible in the CLR; an integer array is not allowed by the CLR to be more than  $2^{31}$  bytes in size, and integers are 4 bytes, so you can't have more than  $2^{29}$  elements in an integer array.

[➔ Reply](#)

20/04/2020

C# Performance Benchmark Mistakes, Part Three - Tech.Pro

http://tech.pro/tutorial/1317/c-performance-benchmark-mistakes-part-three

Go

SEPDEC

MAY

07

13 captures


13 Jul 2013 - 3 Mar 2019

201220132014

About this capture

The midpoint calculation should be  $leftBound + (rightBound - leftBound) / 2$  instead.

↗ Reply




Anderson Milham

posted 6 years ago

Hi Eric,

I really like your posts and I'm always visiting your blog! Just a comment about your code...you forgot to use the "arr2" variable. I know it was a simple mistake and your idea is perfectly clear but it could generate a very wrong result, so I decided to point that.



Eric Lippert

6 years ago

Thanks, I've corrected the typo.

FEED

Tutorials

Blogs

Links

Q&A

Invite Others

PEOPLE

Profiles

Authors

Projects

My Network

MORE

How TechPro Works

About Us

Contact Us

Site Map

Blog

Twitter

Linkedin

Facebook

Copyright © 2013 Tech.Pro