



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 & 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

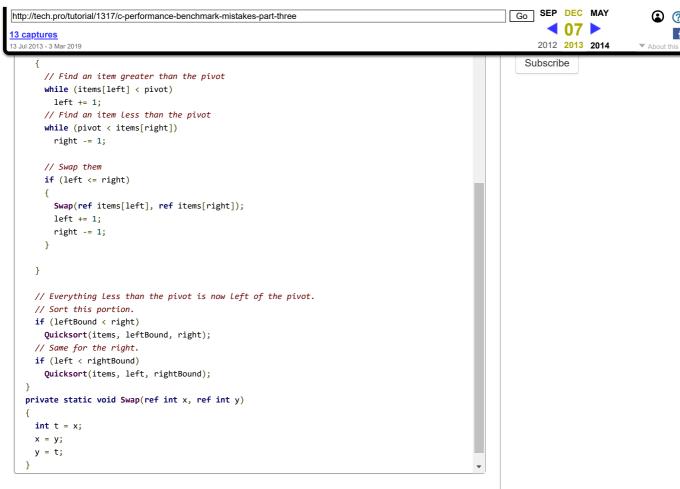


Dear Tech Professional,

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



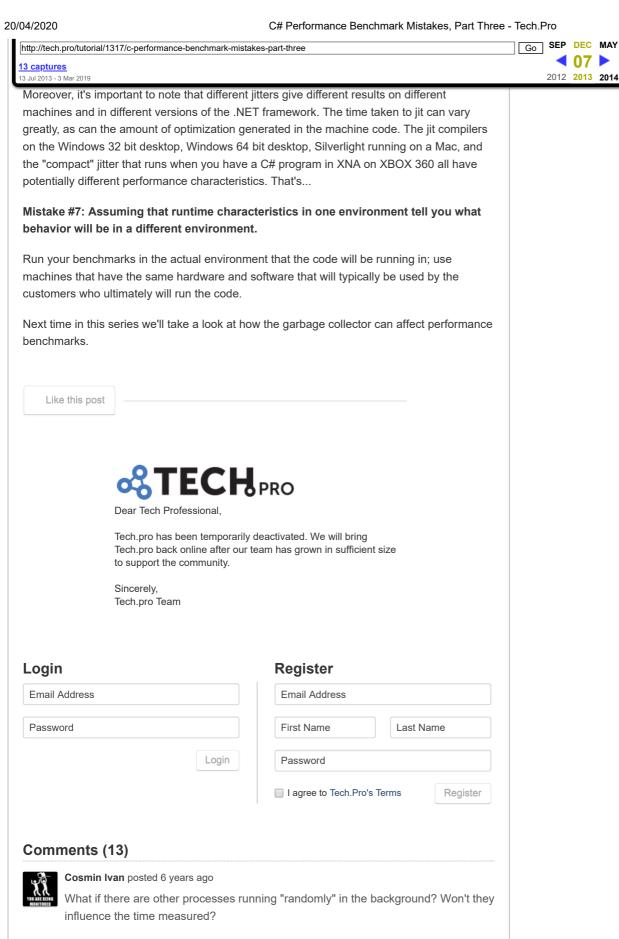
That's a pretty standard implementation of quicksort in C#. Now let's make some modifications to our benchmark test of last time. Here we'll make a random list, copy it to an array twice so that we know that both runs will be operating on exactly the same data, and see if we can deduce the cost of jitting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

◀ 07 ▶





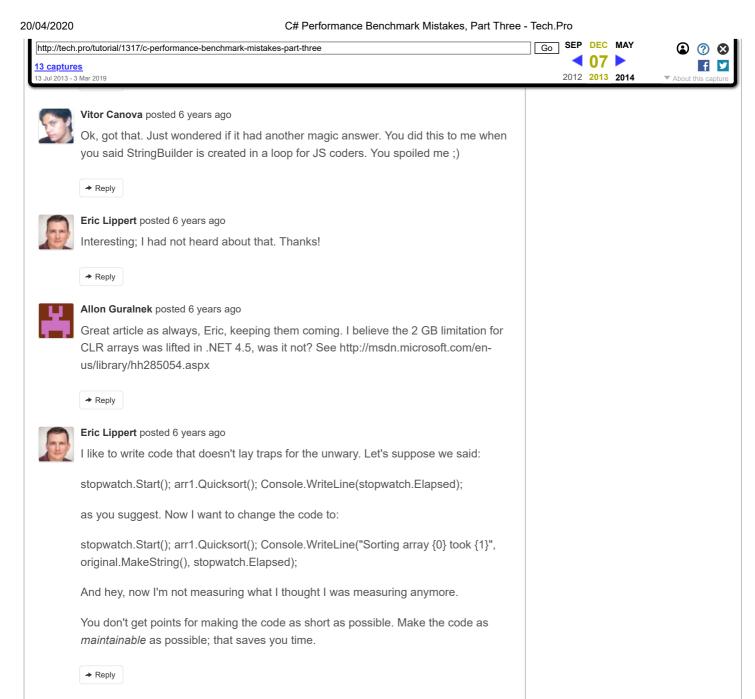
Eric Lippert 6 years ago

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





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

