

# Benchmarking small code samples in C#, can this implementation be improved?

Asked 15 years, 6 months ago   Modified 8 years, 3 months ago   Viewed 27k times



Quite often on SO I find myself benchmarking small chunks of code to see which implementation is fastest.

116



Quite often I see comments that benchmarking code does not take into account jitting or the garbage collector.



I have the following simple benchmarking function which I have slowly evolved:



```
static void Profile(string description, int iterations, Action func) {  
    // warm up  
    func();  
    // clean up  
    GC.Collect();  
  
    var watch = new Stopwatch();  
    watch.Start();  
    for (int i = 0; i < iterations; i++) {  
        func();  
    }  
    watch.Stop();  
    Console.Write(description);  
    Console.WriteLine(" Time Elapsed {0} ms", watch.ElapsedMilliseconds);  
}
```

Usage:

```
Profile("a descriptions", how_many_iterations_to_run, () =>  
{  
    // ... code being profiled  
});
```

Does this implementation have any flaws? Is it good enough to show that implementation X is faster than implementation Y over Z iterations? Can you think of any ways you would improve this?

**EDIT** Its pretty clear that a time based approach (as opposed to iterations), is preferred, does anyone have any implementations where the time checks do not impact performance?

c# .net performance profiling

Share

Improve this question

Follow

edited May 21, 2013 at 15:33



Artemix

2,161 ● 2 ● 23 ● 35

asked Jun 26, 2009 at 3:50



Sam Saffron

131k ● 81 ● 333 ● 510

See also [BenchmarkDotNet](#). – Ben Hutchison Jul 25, 2020 at 1:43

## 11 Answers

Sorted by: Highest score (default) ▾



Here is the modified function: as recommended by the community, feel free to amend this its a community wiki.

104



```
static double Profile(string description, int iterations, Action func) {
    //Run at highest priority to minimize fluctuations caused by other
    processes/threads
    Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;
    Thread.CurrentThread.Priority = ThreadPriority.Highest;

    // warm up
    func();

    var watch = new Stopwatch();

    // clean up
    GC.Collect();
    GC.WaitForPendingFinalizers();
    GC.Collect();

    watch.Start();
    for (int i = 0; i < iterations; i++) {
        func();
    }
    watch.Stop();
    Console.Write(description);
    Console.WriteLine(" Time Elapsed {0} ms", watch.Elapsed.TotalMilliseconds);
    return watch.Elapsed.TotalMilliseconds;
}
```

Make sure you **compile in Release with optimizations enabled, and run the tests outside of Visual Studio**. This last part is important because the JIT stints its optimizations with a debugger attached, even in Release mode.

Share

Improve this answer

Follow

edited Sep 3, 2016 at 23:27

community wiki

6 revs, 6 users 70%

Sam Saffron

You might want to unroll the loop by some number of times, like 10, to minimize the loop overhead. – Mike Dunlavey Jun 26, 2009 at 12:49

3 I just updated to use `Stopwatch.StartNew`. Not a functional change, but saves one line of code. – [LukeH](#) Jun 26, 2009 at 13:23

1 @Luke, great change (I wish I could +1 it). @Mike im not sure, i suspect the virtualcall overhead will be much higher than the comparison and assignment, so the performance diff will be negligible – [Sam Saffron](#) Jun 27, 2009 at 0:05

2 What do you think about showing the average time. Something like this: `Console.WriteLine("Average Time Elapsed {0} ms", watch.ElapsedMilliseconds / iterations);` – [rudimenter](#) Jun 28, 2012 at 13:07

1 I wrote a blog post explaining why this sample doesn't work in *some* situations, see [mattwarren.org/2014/09/19/the-art-of-benchmarking](http://mattwarren.org/2014/09/19/the-art-of-benchmarking) – [Matt Warren](#) Sep 20, 2014 at 21:57



24

Finalisation won't necessarily be completed before `GC.Collect` returns. The finalisation is queued and then run on a separate thread. This thread could still be active during your tests, affecting the results.



If you want to ensure that finalisation has completed before starting your tests then you might want to call [GC.WaitForPendingFinalizers](#), which will block until the finalisation queue is cleared:



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

Share Improve this answer Follow

answered Jun 26, 2009 at 9:50



[LukeH](#)

269k ● 59 ● 370 ● 411

11 Why `GC.Collect()` once more? – [colinfang](#) May 6, 2013 at 19:31

8 @colinfang Because objects being "finalized" are not GC'ed by the finalizer. So the second `Collect` is there to make sure the "finalized" objects are also collected. – [MAV](#) Jul 28, 2014 at 23:50



16

If you want to take GC interactions out of the equation, you may want to run your 'warm up' call *after* the `GC.Collect` call, not before. That way you know .NET will already have enough memory allocated from the OS for the working set of your function.



Keep in mind that you're making a non-inlined method call for each iteration, so make sure you compare the things you're testing to an empty body. You'll also have to accept that you can only reliably time things that are several times longer than a method call.



Also, depending on what kind of stuff you're profiling, you may want to do your timing based running for a certain amount of time rather than for a certain number of iterations -- it can tend to lead to more easily-comparable numbers without having to have a very short run for the best implementation and/or a very long one for the worst.

Share Improve this answer Follow

answered Jun 26, 2009 at 4:18



Jonathan Rupp

15.8k ● 5 ● 48 ● 61

- 
- 1 good points, would you have a time based implementation in mind? – [Sam Saffron](#) Jun 26, 2009 at 5:51
- 



7



I think the most difficult problem to overcome with benchmarking methods like this is accounting for edge cases and the unexpected. For example - "How do the two code snippets work under high CPU load/network usage/disk thrashing/etc." They're great for basic logic checks to see if a particular algorithm works *significantly* faster than another. But to properly test most code performance you'd have to create a test that measures the specific bottlenecks of that particular code.

I'd still say that testing small blocks of code often has little return on investment and can encourage using overly complex code instead of simple maintainable code. Writing clear code that other developers, or myself 6 months down the line, can understand quickly will have more performance benefits than highly optimized code.

Share Improve this answer Follow

answered Jun 26, 2009 at 4:22



Paul Alexander

32.3k ● 16 ● 99 ● 151

- 
- 1 significant is one of those terms that is really loaded. sometimes having an implementation that is 20% faster is significant, sometimes it has to be 100 times faster to be significant. Agree with you on clarity see: [stackoverflow.com/questions/1018407/...](http://stackoverflow.com/questions/1018407/) – [Sam Saffron](#) Jun 26, 2009 at 5:54
- 

In this case significant isn't all that loaded. You're comparing one or more concurrent implementations and if the difference in performance of those two implementations isn't statistically significant it's not worth committing to the more complex method. – [Paul Alexander](#) Jun 28, 2012 at 4:02

---



6

I'd avoid passing the delegate at all:

1. Delegate call is ~ virtual method call. Not cheap: ~ 25% of smallest memory allocation in .NET. If you're interested in details, see [e.g. this link](#).



2. Anonymous delegates may lead to usage of closures, that you won't even notice. Again, accessing closure fields is noticeably than e.g. accessing a variable on the stack.



An example code leading to closure usage:

```
public void Test()
{
    int someNumber = 1;
    Profiler.Profile("Closure access", 1000000,
        () => someNumber + someNumber);
}
```

If you're not aware about closures, take a look at this method in .NET Reflector.

Share Improve this answer Follow

answered Jun 26, 2009 at 4:19



Alex Yakunin

6,598 ● 3 ● 35 ● 52

Interesting points, but how would you create a re-usable Profile() method if you don't pass a delegate? Are there other ways to pass arbitrary code to a method? – Ash Nov 7, 2009 at 7:54

- 1 We use "using (new Measurement(...)) { ... measured code ... }". So we get Measurement object implementing IDisposable instead of passing the delegate. See [code.google.com/p/dataobjectsdotnet/source/browse/Xtensive.Core/...](http://code.google.com/p/dataobjectsdotnet/source/browse/Xtensive.Core/...) – Alex Yakunin Nov 9, 2009 at 12:30

This won't lead to any issues with closures. – Alex Yakunin Nov 9, 2009 at 12:30

- 3 @AlexYakunin: your link appears to be broken. Could you include the code for the Measurement class in your answer? I suspect that no matter how you implement it, you'll not be able to run the code to be profiled multiple times with this IDisposable approach. However, it is indeed very useful in situations where you want to measure how different parts of a complex (intertwined) application are performing, so long as you keep in mind that the measurements might be inaccurate, and inconsistent when ran at different times. I'm using the same approach in most of my projects. – ShdNx Jun 30, 2012 at 17:37

- 1 The requirement to run performance test several times is really important (warm-up + multiple measurements), so I switched to an approach with delegate as well. Moreover, if you don't use closures, delegate invocation is faster then interface method call in case with `IDisposable`. – Alex Yakunin Aug 13, 2012 at 0:38



I'd call `func()` several times for the warm-up, not just one.

5

Share Improve this answer Follow

answered Jul 22, 2009 at 5:15



Alexey Romanov

171k ● 37 ● 322 ● 502





- 1 The intention was to ensure jit compilation is performed, what advantage do you get from calling func multiple times prior to measurement? – [Sam Saffron](#) Jul 22, 2009 at 5:17
- 3 To give the JIT a chance to improve its first results. – [Alexey Romanov](#) Jul 22, 2009 at 5:55
- 1 the .NET JIT doesn't improve it's results over time (like the Java one does). It only converts a method from IL to Assembly once, on the first call. – [Matt Warren](#) Sep 20, 2014 at 21:58



## Suggestions for improvement

4



1. Detecting if the execution environment is good for benchmarking (such as detecting if a debugger is attached or if jit optimization is disabled which would result in incorrect measurements).
2. Measuring parts of the code independently (to see exactly where the bottleneck is).
3. Comparing different versions/components/chunks of code (In your first sentence you say '... benchmarking small chunks of code to see which implementation is fastest.').

### Regarding #1:

- To detect if a debugger is attached, read the property `System.Diagnostics.Debugger.IsAttached` (Remember to also handle the case where the debugger is initially not attached, but is attached after some time).
- To detect if jit optimization is disabled, read property `DebuggableAttribute.IsJITOptimizerDisabled` of the relevant assemblies:

```
private bool IsJitOptimizerDisabled(Assembly assembly)
{
    return assembly.GetCustomAttributes(typeof (DebuggableAttribute), false)
        .Select(customAttribute => (DebuggableAttribute) customAttribute)
        .Any(attribute => attribute.IsJITOptimizerDisabled);
}
```

### Regarding #2:

This can be done in many ways. One way is to allow several delegates to be supplied and then measure those delegates individually.

### Regarding #3:

This could also be done in many ways, and different use-cases would demand very different solutions. If the benchmark is invoked manually, then writing to the console might be fine. However if the benchmark is performed automatically by the build system, then writing to the console is probably not so fine.

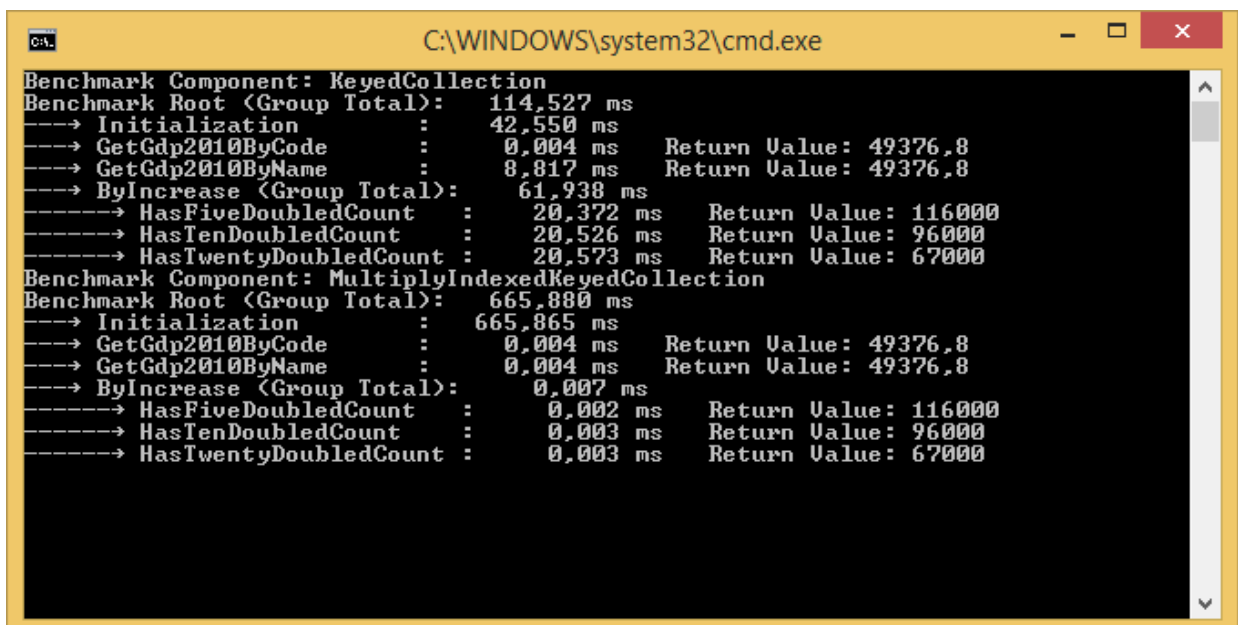
One way to do this is to return the benchmark result as a strongly typed object that can easily be consumed in different contexts.

---

## Etimo.Benchmarks

Another approach is to use an existing component to perform the benchmarks. Actually, at my company we decided to release our benchmark tool to public domain. At it's core, it manages the garbage collector, jitter, warmups etc, just like some of the other answers here suggest. It also has the three features I suggested above. It manages several of the issues discussed in [Eric Lippert blog](#).

This is an example output where two components are compared and the results are written to the console. In this case the two components compared are called 'KeyedCollection' and 'MultiplyIndexedKeyedCollection':



```
C:\WINDOWS\system32\cmd.exe

Benchmark Component: KeyedCollection
Benchmark Root <Group Total>: 114,527 ms
-----> Initialization : 42,550 ms
-----> GetGdp2010ByCode : 0,004 ms Return Value: 49376,8
-----> GetGdp2010ByName : 8,817 ms Return Value: 49376,8
-----> ByIncrease <Group Total>: 61,938 ms
-----> HasFiveDoubledCount : 20,372 ms Return Value: 116000
-----> HasTenDoubledCount : 20,526 ms Return Value: 96000
-----> HasTwentyDoubledCount : 20,573 ms Return Value: 67000
Benchmark Component: MultiplyIndexedKeyedCollection
Benchmark Root <Group Total>: 665,880 ms
-----> Initialization : 665,865 ms
-----> GetGdp2010ByCode : 0,004 ms Return Value: 49376,8
-----> GetGdp2010ByName : 0,004 ms Return Value: 49376,8
-----> ByIncrease <Group Total>: 0,007 ms
-----> HasFiveDoubledCount : 0,002 ms Return Value: 116000
-----> HasTenDoubledCount : 0,003 ms Return Value: 96000
-----> HasTwentyDoubledCount : 0,003 ms Return Value: 67000
```

There is a [NuGet package](#), a [sample NuGet package](#) and the source code is available at [GitHub](#). There is also a [blog post](#).

If you're in a hurry, I suggest you get the sample package and simply modify the sample delegates as needed. If you're not in a hurry, it might be a good idea to read the blog post to understand the details.

edited Sep 16, 2014 at 18:03



Eric Lippert

659k ● 183 ● 1.3k ● 2.1k

answered Sep 14, 2014 at 17:57



Joakim

101 ● 4

Share

Improve this answer

Follow



You must also run a "warm up" pass prior to actual measurement to exclude the time JIT compiler spends on jitting your code.

1

Share Improve this answer Follow

answered Jun 26, 2009 at 4:22



Alex Yakunin

6,598 ● 3 ● 35 ● 52



it is performed prior to measurement – [Sam Saffron](#) Jun 26, 2009 at 5:54



Depending on the code you are benchmarking and the platform it runs on, you may need to account for [how code alignment affects performance](#). To do so would probably require a outer wrapper that ran the test multiple times (in separate app domains or processes?), some of the times first calling "padding code" to force it to be JIT compiled, so as to cause the code being benchmarked to be aligned differently. A complete test result would give the best-case and worst-case timings for the various code alignments.

1



Share

edited May 23, 2017 at 12:34

answered Jun 27, 2012 at 2:16

Improve this answer



Community Bot

1 ● 1



Edward Brey

41.6k ● 21 ● 209 ● 265

Follow



If you're trying to eliminate Garbage Collection impact from the benchmark complete, is it worth setting `GCSettings.LatencyMode` ?

1



If not, and you want the impact of garbage created in `func` to be part of the benchmark, then shouldn't you also force collection at the end of the test (inside the timer)?



Share Improve this answer Follow

answered Sep 15, 2014 at 15:35



Danny Tuppeny

42.2k ● 25 ● 161 ● 286



The basic problem with your question is the assumption that a single measurement can answer all your questions. You need to measure multiple times to get an effective



0 picture of the situation and especially in a garbage collected language like C#.

▼ Another answer gives an okay way of measuring the basic performance.



```
static void Profile(string description, int iterations, Action func) {
    // warm up
    func();

    var watch = new Stopwatch();

    // clean up
    GC.Collect();
    GC.WaitForPendingFinalizers();
    GC.Collect();

    watch.Start();
    for (int i = 0; i < iterations; i++) {
        func();
    }
    watch.Stop();
    Console.Write(description);
    Console.WriteLine(" Time Elapsed {0} ms", watch.Elapsed.TotalMilliseconds);
}
```

However, this single measurement does not account for garbage collection. A proper profile additionally accounts for the worst case performance of garbage collection spread out over many calls (this number is sort of useless as the VM can terminate without ever collecting left over garbage but is still useful for comparing two different implementations of `func`.)

```
static void ProfileGarbageMany(string description, int iterations, Action func)
{
    // warm up
    func();

    var watch = new Stopwatch();

    // clean up
    GC.Collect();
    GC.WaitForPendingFinalizers();
    GC.Collect();

    watch.Start();
    for (int i = 0; i < iterations; i++) {
        func();
    }
    GC.Collect();
    GC.WaitForPendingFinalizers();
    GC.Collect();

    watch.Stop();
    Console.Write(description);
    Console.WriteLine(" Time Elapsed {0} ms", watch.Elapsed.TotalMilliseconds);
}
```

And one might also want to measure the worst case performance of garbage collection for a method that is only called once.

```
static void ProfileGarbage(string description, int iterations, Action func) {  
    // warm up  
    func();  
  
    var watch = new Stopwatch();  
  
    // clean up  
    GC.Collect();  
    GC.WaitForPendingFinalizers();  
    GC.Collect();  
  
    watch.Start();  
    for (int i = 0; i < iterations; i++) {  
        func();  
  
        GC.Collect();  
        GC.WaitForPendingFinalizers();  
        GC.Collect();  
    }  
    watch.Stop();  
    Console.WriteLine(description);  
    Console.WriteLine(" Time Elapsed {0} ms", watch.Elapsed.TotalMilliseconds);  
}
```

But more important than recommending any specific possible additional measurements to profile is the idea that one should measure multiple different statistics and not just one kind of statistic.

[Share](#) [Improve this answer](#) [Follow](#)

answered Sep 19, 2014 at 22:08



[Ms. Molly Stewart-Gallus](#)

1,168 ● 10 ● 25